Analysis and Design of Application Scenarios

G. Blair, K. Cheverst, H. Duran-Limon, A. Friday,
G. Samartzidis, T. Sivaharan, P. Sousa and M. Wu

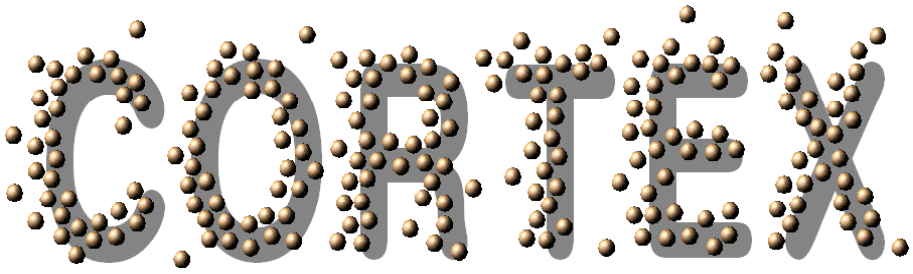
DI-FCUL TR–03–21

July 2003

Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa
Campo Grande, 1700 Lisboa
Portugal

Project IST-2000-26031

CO-operating Real-time sentient objects:
Architecture and Experimental evaluation



Analysis and Design of Application Scenarios

CORTEX deliverable D8

Version 1.0

May 5, 2003

Revisions

Rev.	Date	Comment
0.1	31/03/2003	Integration of partner contributions
0.2	14/04/2003	Second Draft
1.0	05/05/2003	Final version

Editor

Hector Duran-Limon, Lancaster University

Contributors

Gordon S. Blair, Lancaster University
Keith Cheverst, Lancaster University
Hector Duran-Limon, Lancaster University
Adrian Friday, Lancaster University
George Samartzidis, Lancaster University
Thirunavukkarasu Sivaharan, Lancaster University
Paulo Sousa, University of Lisboa
Maomao WU, Lancaster University

Address

Computing Department,
Lancaster University, Bailrigg,
Lancaster LA1 4YR, UK

Executive Summary

The main goals of this document are twofold. Firstly, we aim to identify appropriate application scenarios for evaluating the CORTEX paradigm. Secondly, we aim to demonstrate integration across project partner's contributions.

More specifically, the main contributions of the document are the following:

- Review of the main characteristics of CORTEX applications
- Description of the demo applications
- Requirement analysis of the demo applications
- Comparative analysis of the demo applications' requirements
- Definition of the middleware architecture
- Software infrastructure for the demo applications
- Hardware infrastructure for the demo applications
- Design of the demo applications
- Summary and overall conclusions

The first part of this documents focuses on identifying suitable demo applications. The WP1-D1 [1] has shown that CORTEX is addressing a diverse range of application types which have different requirements. In particular, it was identified that the covered application scenarios require different degrees of autonomy, consistency and cooperation. However, these applications will commonly involve only a subset of such characteristics. Hence, we have selected two demo applications which altogether cover all the main CORTEX characteristics. The control car scenario focuses on real-time ad hoc environments whereby cars are able to operate automatically to avoid collisions. Complementary, the smart room application pays more attention to the intelligent behaviour of sentient objects in which the conditions of the room are automatically set according to the preferences of the people present in the room.

The second part of the document presents a middleware architecture whereby different partner's contribution are integrated. Such platform makes also possible to reuse the system architecture in diverse applications. More concretely, the architecture will be used for the support of the two demo applications. Moreover, the middleware allows us to provide configuration and reconfiguration support to achieve adaptation when unexpected changes are introduced to the environment. The middleware is basically conformed by a number of component frameworks (CFs). The publish/subscribe CF realises the CORTEX event model [2]. The functionality of the control engine of a sentient object is provided by the context CF. Services can be dynamically discovered by the use of the service discovery CF [3]. Facilities for multicast in ad hoc environments are then provided by the multicast CF. The TCB facilitates the detection of timing failures and support for QoS adaptation is provided by the coverage CF [4]. Lastly, the resource management CF controls the resources used by all the CFs [4]. Finally, the software and hardware infrastructure support for the demo applications is introduced. The design of the two demo applications is presented as well and an overall summary and some concluding remarks are drawn.

Related Documents:

[1] CORTEX. "Preliminary Definition of CORTEX programming Model."CORTEX deliverable D2 version 1.0, March 2002.
[2] Duran-Limon, H. A., G. S. Blair, et al. "Resource Management for the Real-Time Support of an Embedded Publish/Subscribe System." *Technical Report MPG-03-02.* 2003.
[3] Paul Grace, G. B. "Interoperating with Services in a Mobile Environmet." *Accepted in Proc. ACM/IFIP International Middleware Conference (Middleware'2003)*, Rio de Janeiro, Brazil. June 2003.
[4] "Preliminary Specification of Basic Services and Protocols."CORTEX Project. IST-2000-26031. Deliverable D5, February 2003.

Table of contents

1. Introduction

The WP1-D1 [1] has identified the typical applications targeted by CORTEX. For this purpose, a number of application scenarios were illustrated whereby the main application characteristics and application requirements were described. Although all scenarios evoke the usefulness of the sentience paradigm, it was shown that CORTEX is addressing a diverse range of application types which have different requirements. In particular, it was identified that the covered application scenarios require different degrees of autonomy, consistency and cooperation.

The main goals of this document are twofold. Firstly, we aim to identify appropriate application scenarios for evaluating the CORTEX paradigm. Secondly, we aim to demonstrate integration across project partner's contributions. The latter is achieved by placing the different contributions in a middleware platform. Such platform makes also possible to reuse the system architecture in diverse applications. More concretely, the architecture will be used for the support of the demo applications. Moreover, the middleware allows us to provide configuration and reconfiguration support to achieve adaptation when unexpected changes are introduced to the environment.

Regarding the application scenarios, the requirements analysis and design of two demo scenarios are presented. The first scenario is an automatic car control system which intends to demonstrate the feasibility of the sentient object paradigm for real-time ad hoc environments. More concretely, we aim at evaluating the event model along with task model and TCB for the support of timeliness and reliability, accordingly. In this scenario, cars are able to operate independently and cooperate with each other to avoid collisions. The scenario will be realised with a number of car robots which are controlled by Pocket PC handheld devices running Windows CE 3.0.

The second scenario focuses on demonstrating the usefulness of the sentient object's control engine in a pervasive environment. More specifically, we aim at evaluating the suitability of both the rule-based system and inference engine for achieving intelligent behaviour. This scenario considers a smart room system in which the intensity of light, room temperature and some other features are automatically tuned according to the preferences of the persons present in the room. This scenario will be deployed in the Innovative Interactions Lab of Lancaster University. The lab counts with an iris scanner, two plasma screens, air-conditioning and DrDAQ sensor boards among other features.

The structure of this document is as follows. Section 2 reviews the main characteristics of CORTEX applications which will then be used for the requirements analysis. A detailed description of the two demo scenarios along with their requirement list is presented in section 3. The degree to which each of the scenarios covers the general CORTEX characteristics is also presented in this section. Section 4 then describes the middleware architecture used for the integration of the project partners' contributions. Details of the technology infrastructure as well as a concise design of the demo scenarios are provided in section 5. Finally, section 6 draws a summary and some conclusion remarks.

2. Characteristics of CORTEX Applications

Over the last few years we have seen the proliferation of embedded mobile systems such as mobile phones and PDAs. Ubiquitous computing is also taking off in which multiple cooperating possibly embedded controllers are used. A new kind of applications can now be envisaged with the emergence of both mobile computing and ubiquitous computing. Applications of such kind are characterised by being largely distributed and proactive, i.e. able to operate without human intervention. A set of further characteristics are also involved such as context awareness as a means to sense the surrounding environment.

This project is examining fundamental issues relating to the support of such applications, including the development of middleware for this domain. The CORTEX approach is based on anonymous and asynchronous event models. Such models are well-suited to ad hoc environments including a large number of autonomous processing units. That is, many-to-many communication scenarios are well supported by the anonymous dissemination of information. In addition, asynchronous communication is adequate in systems whereby frequent disconnection is likely to happen as blocking conditions are avoided. Further requirements include support for mobility and non-functional properties such as timeliness and reliability since some of these applications are time-critical. In addition, this middleware must provide support for scalability as pervasive systems require. Importantly, support for evolution must allow for the introduction of new technological developments. More specifically, the middleware is intended to cope with applications including some or all of the following characteristics:

- Sentience.- These applications are context aware so that they posse the ability to perceive the surrounding environment. That is, context information collected from possibly multiple sensors is both interpreted and analysed.
- Autonomy.- Applications are proactive, i.e. able to operate without human intervention. Autonomous decision-making is supported. For this purpose, some kind of intelligent behaviour should be provided.
- Cooperation.- The constituent sentient objects of an application are able to interact between each other to achieve common goals.
- Large scale.- Applications operate in a pervasive environment whereby a large number of hardware and software components are typically involved.
- Time criticality.- This kind of applications are able to provide hard and/or soft real-time guarantees and dependability assurances. That is, in case of unexpected changes introduced to the environment these applications are able to dynamically adapt to meet their deadlines. In the worse case scenario, the applications are taken to a safe state in case of a timing failure detection.
- Safety criticality.- Rgards applications that may put at risk the lives of human beings. For this purpose, these applications support dependability assurances. That is, the applications are taken to a safe state in case of a timing failure detection.
- Geographical dispersion.- Involve applications which are largely dispersed. The rage of geographical dispersion may vary from buildings and cities to the level of countries and continents.
- Mobility.- These applications involve the physical mobility of hardware components along with their associated hardware components. Mobility may take place in both a fixed and ad hoc environment.
- Evolution.- Regards the case when the growth of the application scales well. In addition, support is provided for both application extensibility and the inclusion of

technology advances. This is achieved by the ability of replacing or adding new components of software and hardware.

The characteristics list outlined above will be used for the requirements analysis of the demo applications. Such analysis will provide all the specific facilities and capabilities that the applications must support, e.g. timeliness guarantees, ad hoc environment support, etc. However, it should be noted that most CORTEX applications typically only exhibit a subset of the characteristics list. Nevertheless, it is clear that all applications will at least be involved, at different levels, with a number of central characteristics. These major characteristics include: sentience and autonomy. That is, it is expected that all possible CORTEX applications will support some degree of context awareness and proactiveness.

The next section presents the definition of the demo applications which includes a detailed description of the applications and the requirements analysis.

3. Definition of Demo Scenarios

3.1 Car Control Scenario

3.1.1 Description of Application Scenario

This demo application envisages future car systems that will be able to transport people without human intervention. That is, the only type of information that will need to be provided is the final destination and optionally the desired time of arrival. Hence, the control car system will automatically select the optimal route according to desired time for reaching the destination, distance, current and predicted traffic, weather conditions, etc. Cars would then cooperate with each other to move safely on the road, reduce traffic conditions and reach their destinations. One of the main challenges that this kind of scenario imposes are those related to time critical systems in ad hoc environments. That is, messages should be timely delivered in geographical areas where cars communicate in ad-hoc environments, where there are wireless bandwidth limitations, disconnections and ad-hoc communications.

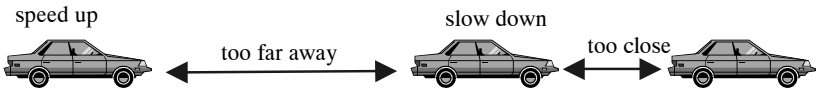


Figure 3.1 Speed control scenario

The main goal for this demo application is to demonstrate the feasibility of the sentient object paradigm for real-time and ad hoc environments. The control system demo application that will be prototyped includes three specific case scenarios:

1) The ability of a car to avoid collisions with:
    a. obstacles
    b. other cars
2) The ability of a car to obey traffic lights
3) Dissemination of non critical data such as sports news, weather and peer to peer information notifications such as traffic updates.

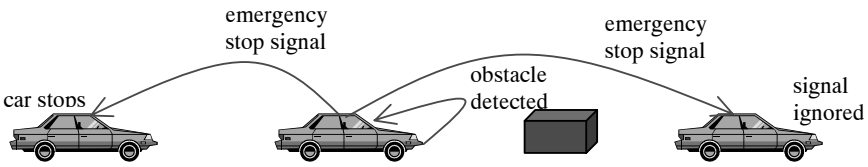


Figure 3.2 Emergency stop scenario

The first case scenario of the system involves the ability to avoid colliding with both obstacles on the road and other cars. For this purpose, cars will be able to detect its position and the other cars' positions and take actions upon this information. The specific characteristics of this case scenario are:

1) slow down / speed up the car speed according to the distance of the car ahead
2) apply an emergency stop if:

a. an obstacle is detected

b. an emergency stop signal is received

c. a failure in the communication system is detected

That is, the application should slow down a car if it is too close to the car ahead as depicted in figure 3.1. Cars, however, should speed up till reaching cruise speed (just below the speed limit) if cars ahead are at a safe distance. An emergency stop should then be applied if a detected obstacle is ahead and too close. Cars applying an emergency stop inform the nearby cars of this. As a result, cars receiving an emergency stop signal from a car ahead should also apply an emergency stop themselves as shown in figure 3.2. Lastly, in case of detecting a failure in the communication system, cars apply an emergency stop to avoid colliding.

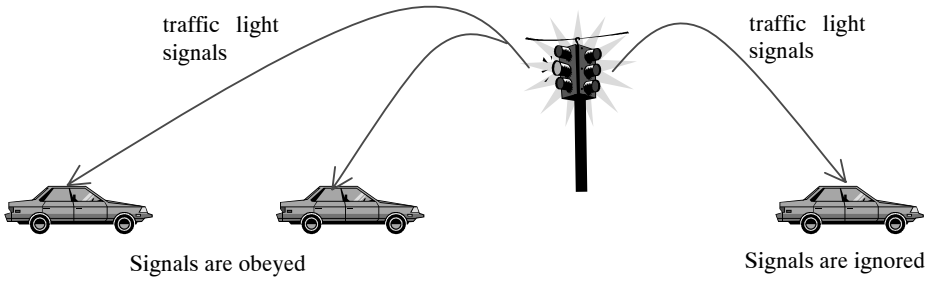


Figure 3.3 Traffic light scenario

The second case scenario, as shown in figure 3.3, refers to the ability to obey traffic lights. Similar to the first case scenario, a car should be able to detect the position of itself, the nearby cars and upcoming traffic lights. Based on this information, cars will be able to act according to traffic light signals. More concretely, the characteristics of this case scenario are:

1) Stop and move according to traffic light signals

2) Ignore traffic light signals when the traffic light has been passed

3) Only one way is assumed

Cars should slow down and eventually stop in the presence of either yellow or red signals. Cars waiting for the green signal will smoothly accelerate till reaching cruise speed when the green signal is received. Importantly, cars that are positioned beyond the traffic light should ignore traffic light signals. In order to simplify the realisation of this case scenario only one way is assumed.

The third case scenario refers to the general non critical tasks. This takes place in the cars such as cars receiving sports news, weather forecast, traffic updates and traffic lights collecting traffic statistics to optimise traffic signal sequence. Most of these tasks will be simple simulations taking place in cars and traffic lights and represent non critical distributed tasks taking place in the car control system. This scenario highlights that the car control system deals with non critical distributed tasks as well, apart from the distributed critical tasks which are the focus of scenario 1 and 2 mentioned above.

3.1.2 Requirements Analysis

3.1.2.1 Sentience

The capability to sense the surrounding environment is a primary requirement for this case scenario. Cars should be able to identify the distance and speed of the cars ahead as well as obstacles in order to avoid collisions. Emergency stop signals should be rapidly published to the cars at risk of colliding. In addition, cars need to be aware of the traffic light signals that are disseminated. Therefore, cars need a number of sensors that will receive both periodic and sporadic events. Examples of periodic events include localisation information broadcasted by nearby cars. On the other hand sporadic events provide information about unpredictable situations such as that of an emergency stop signal.

Capabilities for interpreting context information are also required as raw sensed data may be meaningless in certain situations. For instance, a speed sensor providing only raw speed data cannot know whether the car ahead is suddenly braking. Therefore, the sensed data needs to be transformed to meaningful information as follows:

Input events : Raw speed data consumed periodically from the speed sensor.

Control Logic : Past and current speed data is captured and stored. A rule-based engine determines whether the car is applying a sudden brake.

Output events : In case of a sudden brake, the car publishes an emergency stop event to notify the nearby cars about this situation.

It is also important to keep a record of past context information. That is, this information can be useful to have a more accurate image of the environment. For instance, a history of the previous positions, expressed in latitude and longitude GPS values, of car *a*, car *b* can provide us information to deduce which car is ahead.

The sensor data contribute in deriving context information about each car. The context information includes the position of each car with respect to other cars and traffic light. For instance, each car needs to be aware of its safety zone and be able to detect any objects entering its safety zone. This context awareness can be derived by exploiting for instance, using ultrasonic sensor readings, which are fitted at each side of the car.

3.1.2.2 Autonomy

Cars need to make autonomous decisions such as deciding when to brake or change the car speed. Decision-making should take place based on current and history context information. In addition, a control logic engine is required to carry out such autonomous behaviour. Moreover, the autonomous decision making of each car is supported by consumption of varied sensor information, deriving context information using sensor data and using context aware rule engine. Autonomous behaviour of cars has to be facilitated by a set of corresponding rules. For example, to ensure each car does not enter the safety zone of other cars, the car should continually sense the surrounding environment for nearby objects and take measures to avoid collisions. One such rule can be ‘slow down/stop, if a car is closely in front of you’. Another rule related to traffic light is ‘stop, if the traffic light signal is red, and go, if it is green’.

3.1.2.3 Cooperation

This case scenario requires some degree of cooperation between cars to avoid collisions. Moreover, all cars in the same zone need to have a consistent global view of the environment. Employing, an anonymous and asynchronous communication model, can be considered as highly suited for this purpose. Cars, notify other cars about the actions it takes by dissemination of events. The events are propagated to all cars in its proximity. Furthermore,

realisation of a consistent global view of the environment can be achieved by real time event dissemination. A real-time publish/subscribe service for the wireless ad-hoc network is suited for this purpose. Consider an example, 'a car suddenly brakes, and the car is followed by many other cars', here the braking car should disseminate an event notifying brake action to cars which follow it, in a timely manner. When the other cars receive the brake event on time, they can take measures to avoid collision of chain of cars. The events should additionally carry context information such as location of braking car for the correct perception of the event.

3.1.2.4 Distribution Scale

This system should be designed to cope in scenarios, where there can be potentially large number of cars, however it should be noted in the demo we use a few cars for practical reasons and emulate large scale characteristics. Therefore, the number of sentient objects interacting will be typically large and varying depending on the congestion on the road. Since each car communicates in a one-to-many fashion without the aide of centralized servers, a group communication mechanism is required to support the underlying communication. The cars and traffic lights must use wireless networks, which use the ad-hoc model for communication. This raises the requirement for ad hoc group communication. Moreover, measures are needed to limit the propagation of events. The ad-hoc group communication is expected to use multi-hop routing protocols because of the limited coverage range of IEEE 802.11b. The multi-hop routing protocols can easily lead to wireless network congestion. Therefore, measures are needed to limit the propagation of events to mobile nodes within the proximity of the producer of event. Moreover, in order for the system to scale, the events should only be routed to consumers who have interest in them and a location aware group communication should be exploited to avoid message propagation beyond the geographical area of interest.

3.1.2.5 Time Criticality

The CORTEX event model needs support for the timely dissemination of events whereby high priority events obtain higher priority channels and higher priority threads. Crucially, support for hard, soft and non real-time is required. It is evident that the control car system should provide hard real-time guarantees for assuring that critical messages, such as an emergency stop signal, have bounded delays. In addition, critical tasks should be provided with CPU guarantees. More relaxed guarantees are required for performing less critical tasks such as gradually incrementing the car speed or receiving the other car's positions when the car is not moving. Lastly, non real-time task involve those that do not impose any timeliness requirements such as receiving weather and traffic congestion information.

Moreover, this involves the need for end to end quality of service in event processing and dissemination based on event types. However, it should be noted that providing absolute guarantees of timely delivery of events in a mobile wireless ad-hoc networks is impossible, given the current state of the art of wireless technology. Nevertheless, mechanisms for dependability should be provided to tackle this situation as considered next.

3.1.2.6 Safety Criticality

In distributed systems based on wireless ad-hoc environments, where an unpredictable number of application components compete for a limited amount of resources such as CPU and communication resource, executions can be affected by unpredictable delays. Since the car control system has strict timeliness requirements, the system is only feasible if timing failure detection and QoS based adaptation is supported.

Cooperation of cars for avoiding collisions involves distributed critical tasks. These tasks must be executed within certain time bounds. A timing failure here can be dangerous. Ensuring timing guarantees in distributed processing in wireless ad hoc networks with large number of mobile nodes is not probable. Therefore a timing failure detection service can be used to detect timing failures, which enables to take fail safe actions upon timing failures. Moreover, the timing failure detection service needs to operate in a dependable way which requires adequate resource reservation for it to operate at each node level (e.g. CPU resource) and predictable access to wireless medium.

3.1.2.7 Geographical Dispersion

The entities involved in the car scenario, i.e. cars and traffic lights, are geographically distributed. Importantly, distributed entities are not fixed but they are highly mobile and communicate in an ad-hoc manner.

Moreover a multi hop routing protocol is required to cover the entire geographical area of interest. Large amounts of events are published, propagated and consumed by various mobile entities (i.e. cars), which can be very taxing on the low bandwidth wireless networks. Thus measures are required to limit the propagation of events by only routing events to mobile nodes which are interested in specific event types. In the cooperating cars case, the mobile nodes have high mobility and the networking topology formed by the mobile nodes are highly dynamic, therefore the multicast protocol should take these factors into account and must be robust in these situations.

3.1.2.8 Mobility

The car control system typically involves an environment with both fixed and ad hoc structure. Elements of the fixed structured include traffic lights, proximity groups (i.e. well defined geographical areas) and the GPS services provided by satellites. However, the car scenario is characterised by a highly mobile environment.

As mentioned before the event service depends on a multicast protocol to route events in ad-hoc networks. In a mobile ad hoc network, nodes move arbitrarily, and due to their limited transmission range, nodes need to cooperate autonomously to route events using multi-hop communication.

Multicast routing based on proactive and reactive ad-hoc routing, using shared state kept in the form of routes and adjacent information, is useful in environments with low mobility. However, in scenarios with high mobility it is unsuitable as shared state and topology information can quickly become outdated. Therefore multicast routing protocols for ad-hoc networks that do not assume shared state/topology provides better support for the car scenario.

3.1.2.9 Evolution

The car control system will have to cope with changing conditions during its lifetime. Moreover, environmental conditions also change at run time. This requires, the system be designed to evolve and the underling support must also be adaptable. Since the scenario represents a heterogeneous environment, there needs to be support for run time adaptation as well. Moreover, the middleware built to support the application, should be useable not only in the specific application scenarios presented earlier, but it should also be re-useable on a much broader application scenarios in the future. Component technology is well recognised as a promising approach to build configurable software systems. By using component technology, one can configure and reconfigure the systems by adding, removing or replacing their constituent components. Significantly components are packaged in a binary form, and can be dynamically deployed within an address space. Additional benefits of component technology

include increased reusability, dynamic extensibility, improved understanding and better support for long term system evolution. Moreover, component technology with reflective features is also required to support deployment time and run time adaptation of the car control system.

3.2. Sentient room demo

Recent years have witnessed advances in the enabling technologies for mobile and pervasive computing, such as increasingly mature end-systems, various kinds of wireless communication protocols, and wireless networking technologies. Mark Weiser's vision of Ubiquitous Computing (UbiComp) seems closer than ever to reality. However, only until such systems are widely deployed and integrated with our everyday lives, will Weiser's vision be fully realised. Recently one of the hot research topics in UbiComp is to build and deploy the "Intelligent Environment" or "Smart Environment". Famous research projects have been carried out through the world, such as MIT's Oxygen [5], AT&T's Sentient Computing [6], Microsoft Research's EasyLiving [7], GIT's Aware Home [8], etc.

The CORTEX project is focused on the concept of the sentient object, which is an intelligent software component that is also able to sense its environment via sensors and react to sensed information via actuators [9]. One of the main challenges of CORTEX is to ensure the timeliness and predictability in dynamic environments by using the sentient object paradigm. While the car demo is intended to achieve both properties, the sentient room demo is more interested in illustrating the details of the internal architecture of sentient objects and their supporting infrastructure. Predictability can be achieved by applying the intelligent room control logic (or rules) to the current context, and the control logic has been learned from the previous contexts and interactions between the users and the environment.

3.2.1 Description of Application Scenario

The applications to be built in this sentient room can be divided into two categories: personalized intelligent services and multi-person triggered actions. Here is a sample scenario for the personalized intelligent services. While Alice enters into the sentient room (assuming that it is a semi-public living room), her identity is captured by some device in the room and it starts to behave intelligently in her preferred way. The room finds that it is too dim for Alice to do a given task under the current light intensity, so it automatically turns on additional lights. It sets the room temperature so that Alice will feel comfortable. When Alice sits on the sofa, the room knows Alice is relaxing herself and does the following according to her preferences: it switches on the TV and the Hi-Fi system, tunes to her favourite music channel, and starts playing. By gathering raw sensory data from embedded "invisible" sensors, the intelligent room can then process and analyze them to deduce high-level context and infer personal preferences. Personal preference is not static, but changes with various situations, e.g., time of the day, location, working or relaxing, etc. By associating the personal preferences with context information can make the system behave more intelligently.

Another category of applications involve more than two people in the room simultaneously. For example, when a group of people are in the room and talking to each other, the room might infer that there is a meeting going on and start reacting according to the meeting scenario, e.g., switching on the plasma screen displaying one of the attendee's slides. Having multiple persons in the room raises another interesting question: how should the room react to resolve the conflicts between different person's preferences? For example, if there are more than two persons in the room, the intelligent room should be able to decide what the temperature to set so that it can minimize the disturbance, what kind of music to play to maximize satisfaction. Though building applications in this scenario, we can further explore the suitability of the sentient objects to autonomously resolve conflicts between different personal preferences.

3.2.2 Requirements Analysis

3.2.2.1 Sentience

The sentient room consists of a series of sensors and actuators, which are actually abstract wrapper objects around driver software for particular hardware devices. Sensors emit CORTEX software events in reaction to a real-world stimulus detected by the physical devices [9]. For example, there is an iris scanner at the entrance of the sentient room, which can be wrapped as a sensor in CORTEX. It publishes CORTEX related software events if someone is entering the room and recognized by the iris scanner device. On the other hand, actuators consume CORTEX related software events, and react by attempting to change the state of the real world in some way via some hardware devices [9]. A good example of an actuator is a wrapper object around the air-conditioner, which consumes CORTEX related software events and changes the room temperature through this conventional room device.

Besides sensors and actuators objects, the most important software entities in the room are modelled as sentient objects. Apart from the sensing and actuating capabilities, sentient objects are also context-aware, autonomous, cooperative, and intelligent. Sentient objects should be aware of both their internal state and the state of their surrounding local environment [9]. Sentient objects can make autonomous decisions by themselves, but they can also cooperate with each other both through traditional communication channels and via the physical environment. Intelligence is also given to the sentient objects through the mean of inference and learning, by which the control logic is either statically programmed or dynamically generated. In the sentient room demo, there can be a large sentient object that maintains the logic for controlling every device in it. But this strategy is not scale very well and is not easy to extend, a better idea is to split this super-large sentient object into a series of small sentient objects, each of which has the logic for controlling a single device. If we regard these sentient objects together as a unit, it is actually a sentient object because it both consumes and produces CORTEX related software events, and lies in some control path between at least one sensor and one actuator [1].

3.2.2.2 Autonomy

Every sentient object should have its own internal control logic and make autonomous decision by itself. Taking the personalized intelligent service for example, we might have a number of small sentient objects supporting each of the services, e.g., personalized air-conditioner, personalized news page displayer, personalized music player, etc. Each of these sentient objects can make autonomous decision regarding their specific services.

Context-Awareness

Sentient objects should be aware of both their internal state and the state of their surrounding local environment, and interact with their environment via sensors and actuators. For example, the sentient room should be able to know the temperature, noise level, light intensity, how many people in the room, etc. Any information sensed from the environment that may be used to describe the situation of a sentient object is defined as the context information for the sentient object model [9].

Three main components have been identified by TCD in order for a sentient object to be context-aware: context acquisition, context representation, and inference. Getting contextual information from raw sensory data is the main task of context acquisition, and the major issues in the area of sensory capture are data filtering and sensor fusion. Dead-reckoning can be used if there is insufficient traffic or delays, and fusion can provide more accurate estimates of context. The context representation component deals with the representation of context information in a way that is useful to the sentient object and may be easily exchanged amongst sentient objects. A possible way of representing the context is to use XML, which is expressive, flexible, standard, and easy to handle. The inference component is actually the brain of a sentient object, and it has some form of decision making ability and intelligence.

Inference

Inference engine has been designed as a part of the sentient object architecture, and the intelligence of a sentient object is actually realized by the inference engine and it associated knowledge base. Inference can be applied to the sentient room demo at two levels. High-level inference can be used for deciding how the sentient room should react according to different situations of the room. For the intelligent personalized services, the room could use the current contexts as the input to the inference engine, which in turn decides what actions to take by searching the rule base and outputting the corresponding rule. Besides, inference can also be applied to get low-level contexts, e.g., inferring how many people is in the sentient room by analyzing the environmental variables, such as temperature, noise level, light intensity, etc.

Learning

Instead of capturing knowledge directly from human experts, the intelligent room can also learn how to react from user's previous behaviours. In this mode, the room control logic (or rules) is no longer statically programmed into the system by the designers, but it can evolve with the interactions between the room and its occupiers.

3.2.2.3 Cooperation

In order to achieve the above personalized intelligent services, there might be some supporting sentient objects that provide common functionalities required by the service sentient objects. For example, there can be a "person in the room" sentient object, which consumes events from environment sensors and produces events to the service sentient object. The events from the sensors can be environmental parameters, e.g., temperature, sound level, light intensity, etc, and the produced event contains information about whether there is person in the room. In this sense, although the sentient objects can make autonomous decisions, they should also be able to cooperate with other sentient objects by event based communication. Here the events need not be limited to software events, but real world events as well. Coordination through the environment or "*stigmergy*" is a highly decentralized method of coordination where individual entities follow the same set of simple behaviour rules to yield robust and adaptive coordination, without the need for expensive and unreliable communication [9]. In the sentient room demo, the personalized air-conditioner service sentient object will work together with the temperature sensor and actuator. Changing room temperature might be a task involving constant monitoring the environment data and signalling the actuator.

3.2.2.4 Distribution Scale

Typically, the sentient room demonstration should accommodate a reasonable number of sensors and actuators, as well as the sentient objects. Since the sentient room is context-aware, it should constantly monitor the contextual status of itself. In order to do that, a large number of software events might be produced and emitted from the different sensor objects. Anonymous event channel communication can significantly reduce the number of messages, and efficiently use the network bandwidth.

3.2.2.5 Time Criticality

This requirement is not applicable to the sentient room demonstration.

3.2.2.6 Safety Criticality

This requirement is not applicable to the sentient room demonstration.

3.2.2.7 Geographical Dispersion

This requirement is not applicable to the sentient room demonstration.

3.2.2.8 Mobility

This requirement is not applicable to the sentient room demonstration.

3.2.2.9 Evolution

Scalability, extensibility, customisability are always the interesting issues in distributed system design. As to the sentient room demonstration, we should always bear in mind the possible introduction of new hardware and applications into the room. Although we are going to build the demonstration in the IIL, it should not prevent us from porting it to other indoor environments.

3.3. Summary and Conclusion

We have presented details of the two demo applications along with their list of requirements. The fundamental characteristics that are typically involved in CORTEX applications have been mentioned. These applications will commonly involve only a subset of such characteristics. In addition, different applications may provide different support level for each of the characteristics being addressed. Importantly, with sentience and autonomy being the most fundamental features of CORTEX applications, at least a certain degree of support for these features is provided by all applications.

We have selected two demo applications for the evaluation of the CORTEX paradigm. As said before, the car control system focuses on time critical systems in ad hoc environments. The smart room system, in contrast, is a scenario in a fixed environment that does not have real-time constraints. Rather, this scenario pays special attention to the issue of intelligent behaviour in pervasive environments.

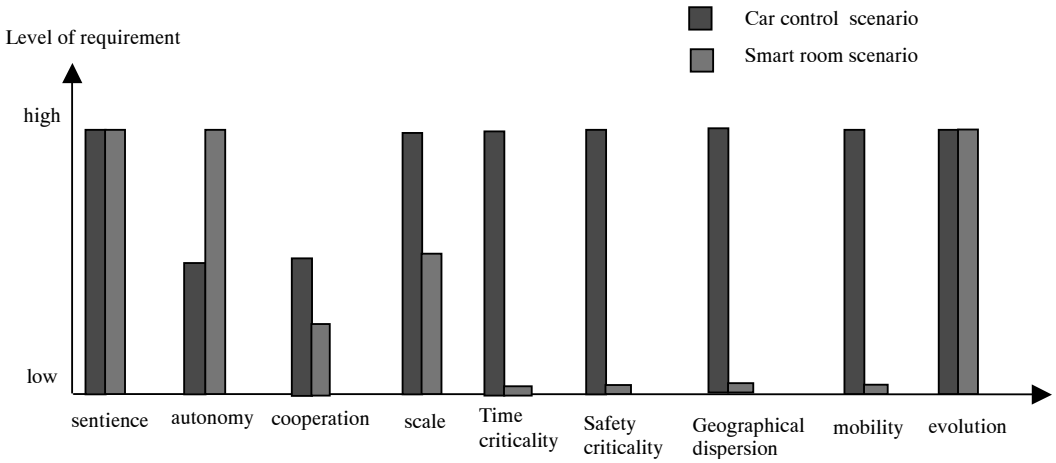


Figure 3.4 Level of requirement for each CORTEX characteristic

A comparative analysis of the characteristics exhibited by the two demo applications is shown in figure 3.4. It can be seen that both applications require a high-level of awareness and evolution. The car system will inevitable need to obtain an accurate image of the environment to achieve the level of coordination that is required. Context information is also essential for the smart room system to infer and tune the room personal preferences. As new technological advances are produced, both applications will require the inclusion of new hardware and software components. As previously stated, the intelligent room demands of a higher degree of both intelligent behaviour and pervasiveness which are reflected on the requirements for autonomy and scale. In contrast, the control car system have much stronger constraints for timeliness and safety criticality. In addition, geographical dispersion and mobility are fundamental requirements of the car scenario whereas in the room scenario the role of these characteristics is negligible. Finally, the smart room system has a lower requirement for coordination as the actions carried out by a sentient object are less likely to have an impact on the other sentient objects. Note that the required levels of coordination of both applications are not high. This is part because of the simplicity of the applications, i.e. more complex car control systems supporting traffic reduction or car overtaking would require a higher level of coordination and cooperation.

We therefore believe that the selected demo applications cover altogether all the characteristics of CORTEX.

4. Background on Middleware Architecture

As identified earlier, the demo applications will be supported by a middleware platform being constructed at Lancaster University. In order to tackle the requirements imposed by ad hoc environments, configuration and reconfiguration capabilities are introduced in the middleware architecture. Based on previous experience in the construction of reflective middleware [10], we make use of *reflection*, *component technology* and *component frameworks (CFs)*. In fact, the implementation of the middleware is developed in OpenCOM [11], which is a lightweight, efficient and reflective model based on Microsoft's COM [12].

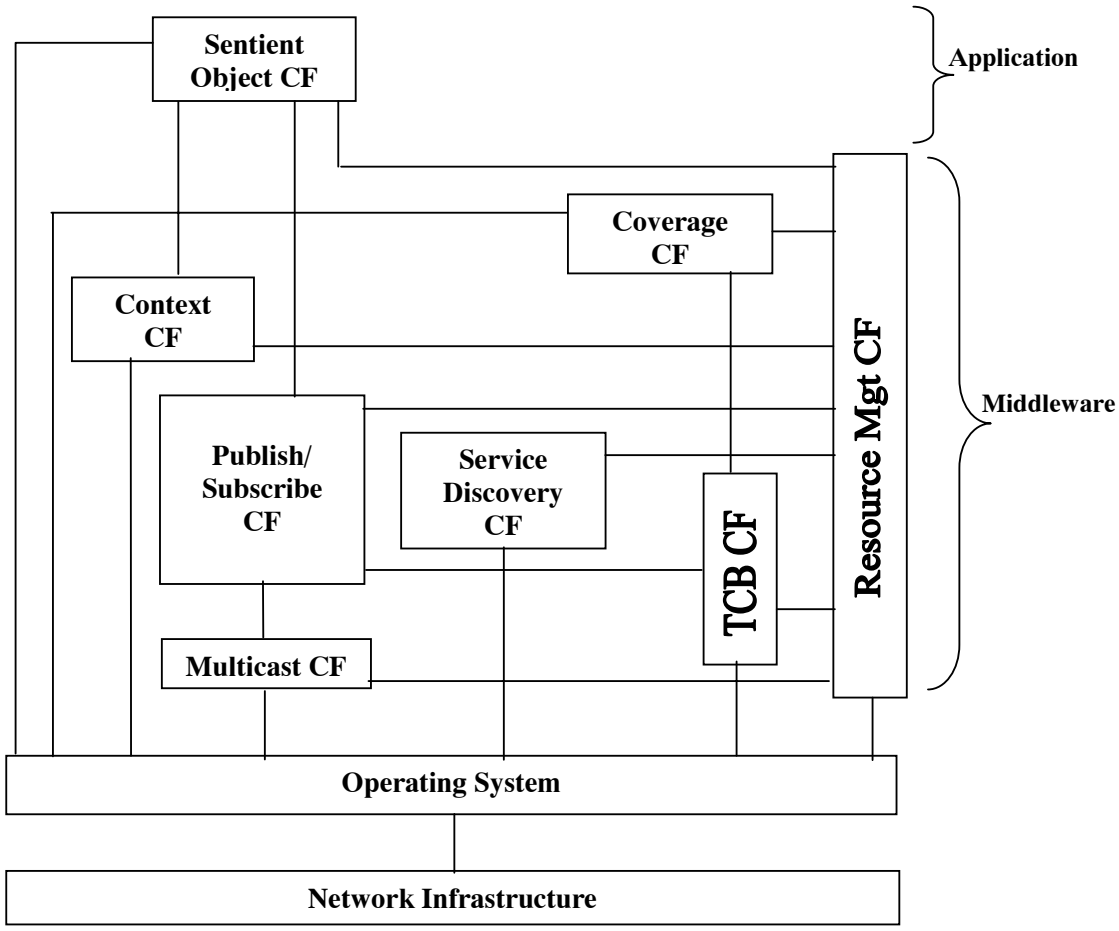


Figure 4.1 The Middleware Architecture of CORTEX

The middleware architecture includes a number of CF as can be seen in figure 4.1. The sentient object CF is in charge of providing the application specific control logic, which is expressed in terms of rules and determines the behaviour of the sentient object. The functionality of handling events and forwarding events to the publish/subscribe system is also provided as shown in figure 4.2. The sentient object CF directly makes use of the context CF for handling context in a uniformed way and publish/subscribe CF to consume and produce events.

The CORTEX event model is realised by a prototype of a STEAM-like [13] publish/subscribe system. Briefly, the publish/subscribe CF follows an implicit event model [13] whereby mediators are not required as entities subscribe to particular event types. This model is suitable for ad hoc environments in which periods of disconnection are unpredictable. Notably, subject-, content- and context-based event filtering is supported. In

addition, filters use a query language (or subscription language), called *Filter Event Language (FEL)* [14]. The event data model exploits XML to represent events. A flexible and general XML profile is defined to represent XML based events. The dissemination of events over the network is then achieved by using SOAP Messaging [15], which is a lightweight XML-based protocol. The SOAPtoMulticast component is in charge of mapping the SOAP messaging protocol to IP multicast. Real-time support, basically including CPU guaranties, is provided by the resource management CF (see below). The API of the publish/subscribe CF is shown in Appendix A.

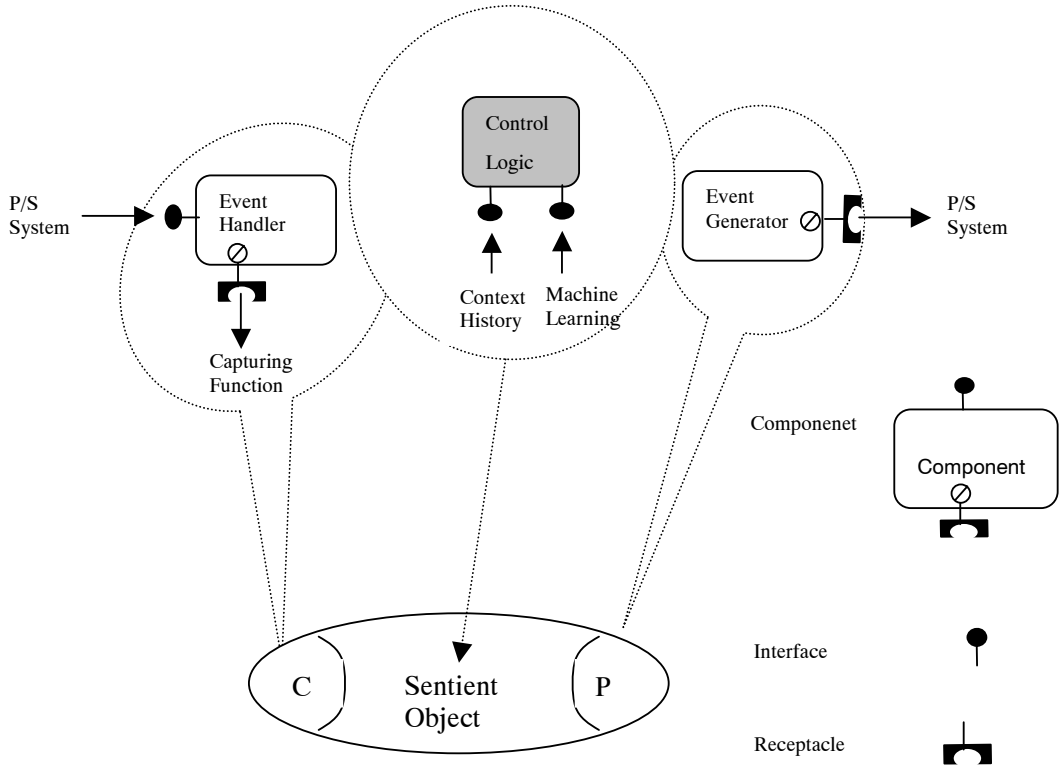


Figure 4.2 Sentient Object CF

The principal function of the Service Discovery framework [3] is to allow services that have been advertised by different service discovery protocols to be discovered. This is performed by changing the component configuration depending on what type of discovery technology is currently used in the environment. For example, if only SLP is currently in use, the framework's configuration will be an SLP Lookup personality. However, if SLP and UPnP are both being utilised at a location then the framework's configuration will include component implementations to discover both. An application may also require services to be advertised; therefore, the personality can be changed to include service registration functionality using one or more protocols of choice. The API of this CF is included in Appendix B.

The main purpose of the resource management CF, previously defined in [4], is to control the resources used by all CFs. For instance, the resource management CF provides some real-time support to the publish/subscribe system. CPU guarantees are supported at this stage. Event types are mapped to tasks and each task have a corresponding VTM [16]. When large number of events are published by various publishing entities for example by using the Publish() method of IPublish interface , the publish method is intercepted using OpenCOM's *method interception* facility and the event type is determined and the rest of the event processing is done by the associated VTM for that task. Similarly, at the subscriber side the Dispatcher component is responsible for ensuring priority based event dispatching. The DispatchEvent method of IDispatcher is intercepted and the priority of the event is determined. The associated VTM is then used for further processing the event, for example filtering of the

event and eventually pushing the event to the consumer. The API of this CF is shown in Appendix C.

The context CF is in charge of providing the functionality of the context management and inference engine used by sentient object. Hence, this CF offers translation functions which transform the data carried by events to meaningful values, as shown in figure 4.3. Moreover, data filtering and sensor fusion are the major issues here: dead-reckoning can be used if there is insufficient traffic or delays, and fusion can provide more accurate estimates of context. Context information is stored in a context history database, and this component provides means to interact with the context history. This database can be shared by multiple sentient objects. An inference engine is in charge of interpreting and executing the rules defined by a sentient object. In addition, machine learning techniques are used to learn new behaviour whereby new rules can be dynamically added to the rule-base of a sentient object. The API of the context CF is presented in section 5.2.2.1.

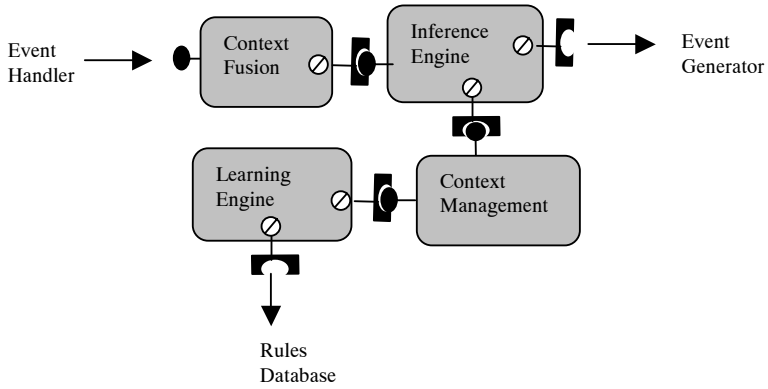


Figure 4.3 Context CF

The multicast CF provides facilities for multicast communication in an ad hoc environment. Research has shown that a single multicast protocol does not typically provides the best solution for all ad hoc environments. That is, these protocols behave differently in high and low mobility conditions. This CF offers facilities for the dynamic replacement of multicast protocols. Currently, an adaptive flooding algorithm based on retransmission probabilities is supported for the dissemination of messages in a high mobility environment. The algorithm tries to maximise the reliability of the multicast (that is, the coverage), without sacrificing too many resources.

The TCB CF [4] provides facilities for the detection of timing failures. In case of the occurrence of a timing failure a specific failure handler is called which will bring the system to a safe state. Duration measurements are also provided in which a duration can be the time taken to transfer a message. In addition, timely execution of sporadic tasks is supported whereby a CPU time slice is guaranteed for their execution. The API of the TCB CF is included in Appendix E. Closely related to this CF is the coverage CF [4] which offers both coverage awareness and coverage stability. The former establishes a coverage function that defines the probability distribution for a given timeliness requirement. Hard real-time systems have associated high coverage probabilities. In case of detecting negative variations on the coverage function, the coverage stability facility gradually adapts the system to maintain the desired coverage. The API of this CF is presented in Appendix D.

Finally, details about the interactions between the CFs are also shown in figure 4.1. The publish/subscribe CF relies on the multicast CF. The sentient object CF uses the publish/subscribe CF to receive and send events. In addition, the TCB provides monitoring functions, adaptation mechanisms and fail-safe procedures to the publish/subscribe CF. Application QoS level adaptation is then supported by the coverage CF which uses the

duration measurement services provided by the TCB. Lastly, the resources used by all CFs are controlled by the resource management CF.

5. Design of Demo Scenarios

5.1 Design of the Car Control Demo

5.1.1 Architecture Overview

This section considers the overall architecture needed to support the car control demo. The motivation to model the entire system based on the basic building block of a primitive sentient object is clear from the requirements analysis. A single car contains a set of primitive sentient objects, which interact among themselves and the environment (other cars and traffic lights). To achieve the objective of cooperation of cars, it is important that the largely distributed sentient objects cooperate autonomously without any central control. Moreover, the sentient objects above need a set of middleware services i.e. publish/subscribe service, resource management service, context awareness and failure detection service.

The design section explores the fundamental integration issues. This will facilitate the construction of large-scale proactive cooperating cars application, composed of sentient objects. The preliminary CORTEX system architecture would form as the model. The proposed CORTEX architecture explicitly considers two logical scopes in terms of the problems that must be dealt. The local scope considers issues that must be dealt within a car. The global scope considers the issues that must be dealt in cooperating large scale autonomous cars, which follow a WAN-of-CAN approach.

In order to materialize the cooperating cars application, it is necessary to define a set of services the middleware components framework should provide. The middleware services are accessed locally by each CORTEX node, but which may have a distributed, global scope. It is important to note that the middleware components are fully distributed and no centralized middleware components exist. This section presents the specification of the basic services needed in the CORTEX middleware to support the car control applications. We focus on how we integrate the constituent services of the CORTEX middleware, in order to fulfill the requirements of the car control application demo. The figure 5.1 shows the architecture overview of the particular configuration of the constituent services of the CORTEX middleware, employed to support the car control application demo.

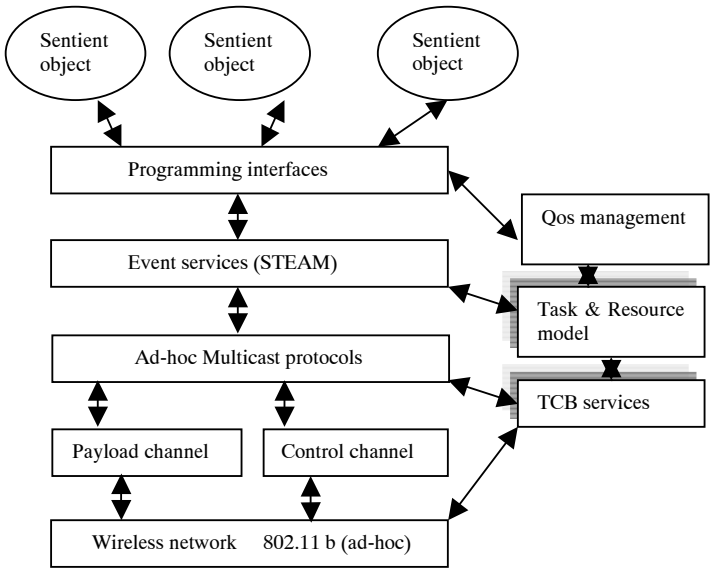


Figure 5.1 Architecture overview

In terms of the structure of this section, we try to follow a top-down approach by motivating the need for the services and by describing and specifying what they should provide to satisfy the requirements of the car control application.

Moreover, the figure 5.1 shows the overall middleware architecture to support the construction of the cooperating cars demo. Next we look at the functional and non functional properties of the middleware components and their role in meeting the applications requirements.

5.1.2 Technology Infrastructure

5.1.2.1 Software Infrastructure

Publish Subscribe Service based on STEAM definition

The cars are mobile nodes and they come into communication in an ad-hoc way. The STEAM [13] definition addresses the issues in building an event service for mobile ad-hoc networks. Thus the car composite sentient objects can utilize the services of a publish/subscribe model based on STEAM definition. The key features of the publish/subscribe service, required to support the car control system is considered next. A prototype of STEAM like publish/subscribe service is designed and implemented using Lancaster universities OpenCOM [10] component technology. OpenCOM is a reflective component model based on Microsoft's COM. More details of OpenCOM can be found in [10].

- ***Event model***

The implicit event model [13] is adopted for the publish/subscribe service because the implicit event model does not assume: fixed infrastructure networks to place event broker(s) and system wide services for the event service to operate. Both of the properties can not be assumed in wireless ad-hoc networks. The implicit event model allows subscribing entities to subscribe to particular types of event and publishing entities to publish events of some type.

- ***Organisation model***

The 'Distributed organisation with collocated middleware' organisation model [13] is employed to architect the publish/subscribe service. This approach facilitates the event middleware to be collocated with the address space of the sentient objects. Moreover this organisation model assumes no form of centralised event middleware components. Furthermore, any centralised middleware components can not be assumed in ad-hoc networks.

- ***Filter constraint language***

A publish/subscribe service employed by the cooperating cars application may support large number of sentient objects, all of which generate events that may contain different types of information. Therefore the number of events propagated in a cooperating car environment may potentially be large. This leads to two requirements: a means for subscriber side of the sentient object to convey their interested in certain categories of events propagated in the system, and a means to control the propagation of events from the publisher side of the sentient object. The subscriber need some subscription language to express the categories of events they are interested. Therefore a filter constraint language, named FEL, is designed to support subject, content and context based event filtering.

- ***Generic event dialect***

Various sentient objects are envisaged to use the publish/subscribe service, where different publishers may want to publish different types of information. On the other hand since publishers and subscribers are anonymous, subscriber too should be able to interpret these events (or notifications) without a priori knowledge. Therefore we have the need for a generic

event dialect which can be understood by all sentient objects in the system. We exploit XML to represent events. The justification for choosing XML to represent events are, we can represent any type of event without being restricted by the event data model, and XML is extensible and XML support interoperability between sentient objects without a priori knowledge of the exact structure of the event. We define a generic XML profile to represent generic events to which all sentient objects conform to. Thus all raw sensory data needs to be translated to this generic XML based event format.

- ***Group Communication (Ad-hoc multicast routing)***

Given the fact that we have chosen the implicit event model and the distributed middleware service with collocated middleware organization model, the natural means of underlying communication mechanism the entities can utilize to communicate is some form of one to many communication patterns. Therefore group communication mechanism is employed as underlying communication mechanism. We exploit multicast group communication. Therefore, we need an ad-hoc multicast routing protocol. More details of the required properties of the multicast protocols are given in the ad-hoc multicast protocols section.

- **Multiple event channels with different functional and non functional properties**

Having a single event channel have the following disadvantages

1. All events (critical and non critical events) published by all sentient objects utilises a single event channel, where there is no mechanism to control the propagation of events from a publisher to all subscribers. This can be highly taxing on the wireless bandwidth.
2. Events get propagated to subscribers who are not interested in those events.
3. Limitations in providing event channels with different quality of service.

Having multiple event channels can overcome the above disadvantages, by having different event channels based on event types. This enables to associate different QoS to event channels.

Event channel :: = < event type , non functional attributes >

Non Functional attributes = {reliability, priority, discard policy, fail safety policy, quality of service based dynamic resource adaptation policy}

For example sudden brake event type, published by cars who suddenly brake, can utilise a real time event channel with high quality of service properties such as timing guarantees(or failure detection and notification), whereas event type related to congestion detection can utilise a non real time event channel(or event cannel with low QoS properties). The event types are mapped to multicast groups. Moreover, no centralised binding service is required to map event types to underlying multicast groups; making it suitable for ad-hoc networks. To build the publish/subscribe service with multiple event channels with associated QoS properties, requires the services of Task and Resource model [16] and Timely Computing Base [17] service, which we look at the Task and Resource model section and TCB section respectively.

- ***Context filtering- Distance based Context***

It is beneficial to support context filtering, in addition to subject and content filtering. For instance, the cars in close proximity need to cooperate and the cars which are not in close proximity need not cooperate. To support this behaviour we need distance based context filtering. Distance context is derived from the location of publishing car node and the location of subscribing car node using GPS coordinates. Moreover, while the subscribers move to different geographical area they may be interested in receiving events which are generated from the new geographical area. With subject and content filters a subscriber is not capable of expressing interest in events as in the above scenario where context comes into play. With context filtering support the subscriber may define a subscription such as “deliver all floating

car data generated from cars located within 100m from me". The distance based context filtering can be considered as an approximation to proximity filters.

Task and Resource model for the support of Real Time Event Service

The most important elements of the resource model are abstract resources, resource factories and resource managers [16]. Abstract resources explicitly represent system resources. In addition, there may be various levels of abstraction in which higher-level resources are constructed on top of lower-level resources. Resource managers are responsible for managing resources, that is, such managers either map or multiplex higher-level resources on top of lower-level resources. Furthermore, *resource schedulers* are a specialization of managers and are in charge of managing processing resources such as threads or virtual processors (or kernel threads). Lastly, the main duty of resource factories is to create abstract resources. For this purpose, higher-level factories make use of lower-level factories to construct higher-level resources. The resource model then consists of three complementary hierarchies corresponding to the main elements of the resource model. Importantly, *virtual task machines* (VTMs) are top-level resource abstractions and they may encompass several kinds of resources (e.g. CPU, memory and network resources) allocated to a particular task.

The publish/subscribe service needs to provide real time event service, by collaborating it with the resource management service (i.e. task and resource model). In the car control application scenario various event types are published and subscribed but the priority of the event types varies. For example sudden brake events published by suddenly braking cars needs high priority in dispatching the event than events published by car which publishes events periodically about its location and speed. The idea being high priority events should be given more priority /resources in processing them than the low priority events to guarantee timely execution and to meet critical deadlines. To meet this requirement event types are mapped to *tasks* and each task has a corresponding VTM [16]. When large number of events are published by various publishing sentient objects, for example by using publish method, the publish method invocation is intercepted, and the event type is examined by the delegate object. Events may carry quality of service parameters which may be considered by the delegate object. The delegate object makes the decision, as to how the event is handled and acts as a task switching point. The rest of the event processing is controlled by the associated VTM for that task. The delegate object takes into consideration in its decision making process the quality of service specifications and the state of the system such as CPU, network load and may exploit feedback information provided by monitoring components of Timely Computing Base [17] about timing failures. Similarly, the subscribing sentient object is responsible for ensuring priority based event dispatching. When the event is received at subscriber side; it is intercepted and the priority of the event is determined by the delegate object and the associated VTM is used for further processing the event. The figure 5.2 shows one possible configuration.

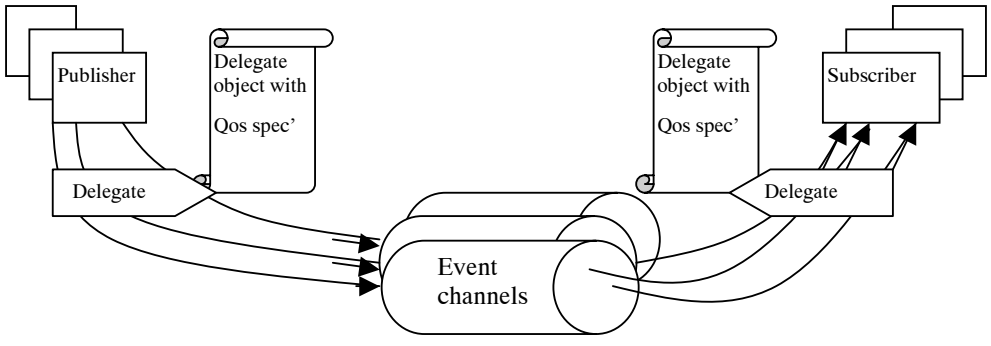


Figure 5.2 Integration of publish/subscribe service & resource and task model

Tables 5.1 and 5.2 show an example of mapping event types, tasks and VTMs .

Event Type	Task
EmergencyStop	CarControl.Critical
Congestion	CarControl.Non-critical

Table 5.1 Example of the Mapping of events type to tasks

Task	VTM
CarControl.Critical	Job-critical
CarControl.Non-critical	Job-Non-critical

Table 5.2 Example of the Mapping of tasks to VTMs

For example when each mobile node has the above mapping, there is a consistent global view of how each node processes event types. For example when emergency stop events are generated and published by the cars, it will be propagated on a specific QoS aware event channel to all interested cars and each consumer dispatches and processes the emergency stop event type with a globally consistent QoS. This forms the first step in achieving distributed critical tasks in a timely manner.

Timely Computing Base (TCB) for the Distributed and Local Timing failure detection

Resources allocation and resource scheduling policies have to be adapted in light of any quality of service violations. The resource and task model has facilities to adapt resource allocations and resource scheduling policies but it additionally needs a feedback mechanism informing it, about any quality of service violations. This additional requirement can be fulfilled by Timely Computing Base (TCB) [17] services. The TCB service informs resource management service about any local and distributed timing failures thus triggering resource and policy adaptations.

In the car control scenario it is quite evident the importance of timely execution of distributed and local tasks. In wireless ad-hoc networks guaranteeing timely dissemination of events from source to sink(s) is difficult mainly because of the dynamically varying contention for the shared wireless medium and limited wireless bandwidth. In the car control application number of cars in a certain area can vary dynamically making the environment highly volatile. In these cases measures should be taken to adapt the system so that critical tasks are not compromised. For example event channels disseminating non critical events can be suspended thus minimizing the load on wireless medium. Thus the ability to detect quality of service violations, can act as a feedback mechanism in triggering adaptation of resources and tasks. Therefore detecting local and distributed timing failure and having appropriate fail safety measures can avoid dangerous consequences. The Timely Computing Base [17] provides the critical timing failure detection services, namely

1. Timing failure detection of timed execution of a local task.
2. Timing failure detection of timed execution of a distributed task.
3. Duration measurement of local tasks with bounded accuracy.
4. Duration measurement of distributed tasks with bounded accuracy.

The publish/subscribe service augmented with task and resource model need to adapt in light of quality of service violations. The TCB services can provide the real time publish /subscribe

service the feedback about quality of service violation (e.g. timing failures). The figure 5.3 shows the model of the feedback mechanism.

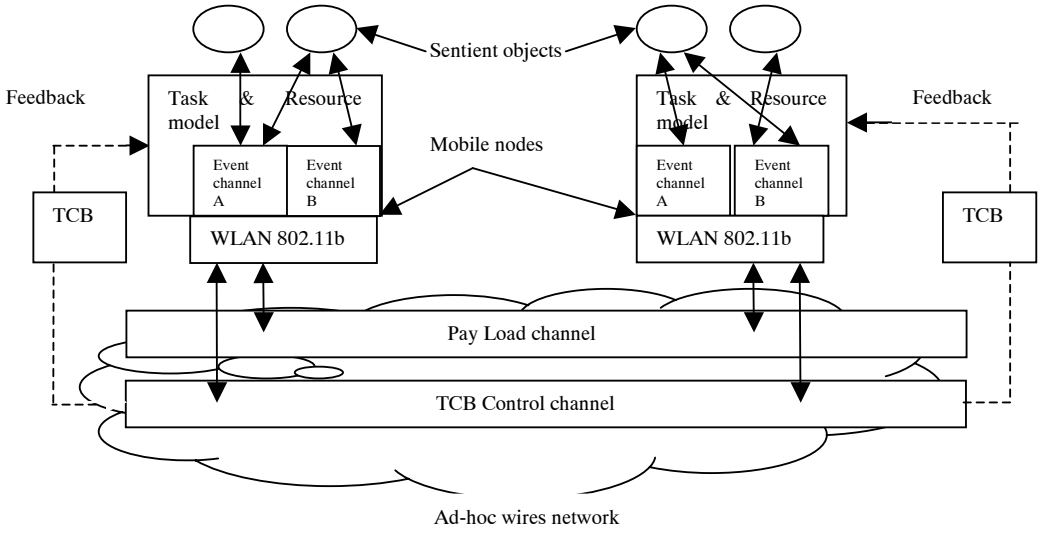


Figure 5.3 Event channel's QoS adaptation based on TCB feedback

The TCB control mechanism works on a dedicated wireless channel separate from the payload wireless channel. TCB service monitors timing failure detection and provides feedback to event channels. The event channels inform the quality of service violations to task and resource management service which re-negotiates the quality of service properties of the respective event channels. The TCB has been ported to Windows CE 3.0. by the University of Lisbon.

Probabilistic Multicast Protocol for Wireless Ad-Hoc Networks

The enabling technology that will allow the dissemination of events between cooperating cars running the publish/subscribe service is a probabilistic flooding multicast protocol. The protocol specifically targets ad-hoc environments where high node mobility and a frequently changing number of group members are present.

The design of the probabilistic multicast protocol is inspired by previous research on multicast algorithms (both proactive and reactive) for ad-hoc networks which points that most existing algorithms (AMRoute, CAMP, MCEDAR, AODV, etc.) perform inadequately when high mobility is present in the environment. The main reason why these protocols fail is due to the fact that they maintain shared state in nodes in the form of routes and adjacent information, which are rapidly outdated due to high node mobility.

For the above reason we have designed a fairly simple multicast protocol based on a probabilistic flooding algorithm with damping, which does not maintain shared state in nodes. The protocol is offered as part of a multicast component framework based on Lancaster's OpenCOM [10] reflective COM technology, which is intended to include more multicast protocols in the future that will cover several environments such as ad-hoc environments with low mobility (where existing multicast protocols can be applied) and wired environments.

Basically, the protocol disseminates multicast messages by flooding them between nodes. This guarantees good performance in high mobility and reliability through redundancy, but of-course it also means that network resources are badly managed. For that reason, two additional mechanisms are employed by the protocol.

The first has to do with probabilistically forwarding flooding messages. That is, each node decides if it should forward a flooding message according to a probability $p \in (0,1]$ which is updated according to the number of duplicates that a node has received from its neighbouring nodes. This effectively minimises the number of unnecessary duplicate messages without sacrificing reliability as we have found experimentally through simulations.

The second mechanism, which is called damping, aims to eliminate the number of unneeded duplicates by allowing nodes to wait for a random; small time interval before they will actually forward a message. During this interval the nodes listen for other neighbouring nodes that will potentially flood the message and if that is the case they forward with a reduced probability.

5.1.2.2 Hardware Infrastructure

Hand-held computers as mobile robotic car platform controllers

The benefits of using small hand-held computers as the controller of robotic car mobile platform are many. Though much smaller than the size of a laptop computer and also having much less capabilities, it is still almost more powerful than current micro-controllers. The advances in hand-held computers make it possible to have mini-computers on smaller mobile robotic car platform than previously possible. Because hand-held computers are stand alone units, they incorporate relatively large quantities of memory, processor power, output display and input output capabilities. Since the size, weight and power requirements of hand-held have been drastically reduced, the use of a mobile hand-held device as the controller of the robotic car is well suited and it could very well be the future of all mobile platforms. The more powerful the processor, the more complex the controlling program can be. The much larger memory allows the use of higher level languages like C/C++ and not just assembly. The built in display provides much easier interface for faster debugging the code and interacting with the programs. Therefore the hand-held computers are used as the mobile robotic car platform controllers.

Hardware setup

The internal architecture of the robotic car is shown in figure 5.4. The pocket PC makes a handy car robotic controller; it packs a lot of computational power in a small size, runs on batteries, and best of all, can display graphics and an interactive user interface. Our car robot empowers a pocket PC to move about and sense the nearby environment autonomously. The base uses four wheels that allow changing driving direction with independent control of rotation, meaning it has a four wheel drive with each wheel attached to a motor. The base also has four ultrasonic sensors (on the four sides of the car) to detect objects in the nearby environment up to about a meter away. The GPS circuit senses the location of the car with an accuracy of approximately 10 m. The actuators (wheels etc) and sensors (GPS, ultrasonic etc) are connected to the pocket PC via a serial synchronization cable. Additional sensors and actuators can be added to the pocket PC, when needed. The pocket PC is equipped with two IEEE 802.11b network cards to enable the robotic car to communicate with other robot cars and traffic lights. One IEEE 802.11b network card is used for the payload channel (event dissemination) and the other for the control channel of the TCB.

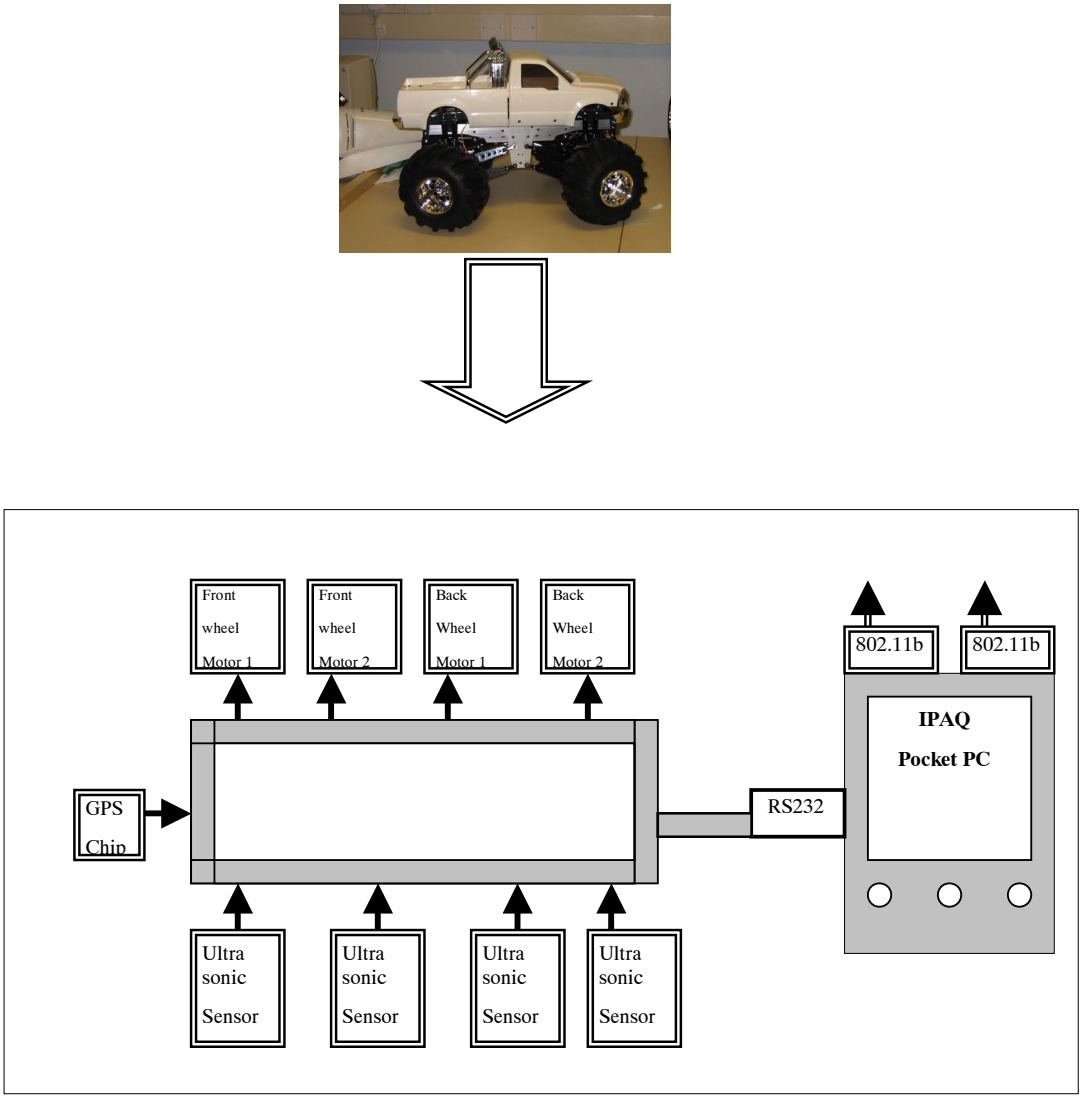


Figure 5.4 Internal architecture of the robotic car

IPAQ Pocket PC

The Pocket PCs used for this purpose are Compaq iPAQ 5450 which incorporate much faster CPU and larger amounts of RAM memory. They are able to run programs compiled for Windows CE.

Figure 5.5 Compaq iPAQ 5450

Features:

Processor	:	Intel® 400 MHz processor with XScale technology
Memory	:	64 MB SDRAM, 48 MB Flash ROM
WLAN	:	1) Inbuilt IEEE 802.11b Support for number of selectable sub channels. Output power of approximately 17dBm (50 mW) 2) External IEEE 802.11b Cisco Aironet350Series PC Card
Operating system	:	Microsoft windows for pocket PC 2002
Expansion capabilities	:	Supports expansion packs with Compact Flash and PC card slots.
Synchronisation	:	Support for USB and Serial

5.1.3 Design of Scenario

Both cars and traffic lights are modeled as sentient objects (SOs). Each car is designed as a composite sentient object that includes a number of primitive sentient objects as shown in figure 5.6. The car composite sentient object can be considered as a mobile node. Mobile nodes communicate by event dissemination via wireless networks (802.11b ad-hoc). Each primitive sentient object inside the car has its own control logic, in other words a context aware ruled based engine. The traffic light will also be a sentient object but won't be as complex as the car composite sentient object. Each primitive sentient object is designed to have a subscriber behavior (consumption of events of interest) and a publisher behavior (publishing of events). Thus, the sentient object requires the services of event based middleware to operate. Each primitive sentient object has the facility to subscribe to events of its interest (using subject, content and context filters). The publisher side of the primitive sentient object publishes events. The publishing of events can be controlled by having publisher side event filters (subject and content based). Each primitive sentient object consumes and publishes different event types. Thus each primitive sentient object can utilize different event channels for event consumption and publishing.

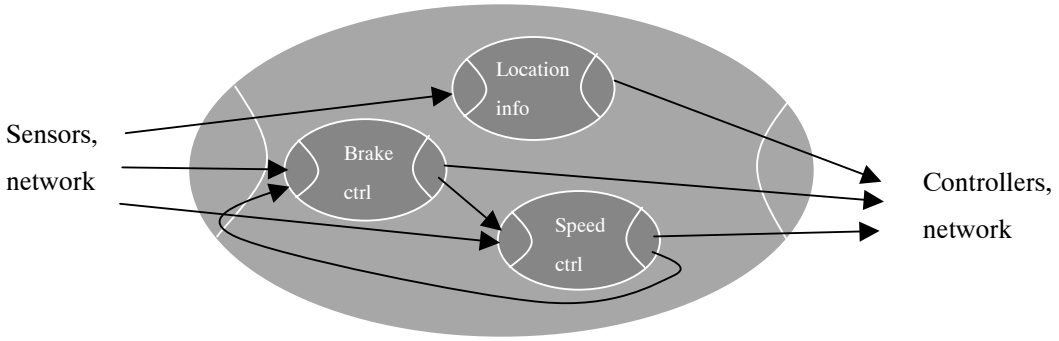


Figure 5.6. Internal Structure of a Car Sentient Object

The input events received by a brake control SO as well as the events produced by this object are shown in table 5.3. Brake control sentient objects receive "emergencyStop" events from other cars. These events are filtered by a distance-based filter of the publish/subscribe system. A history of the previous cars' positions is kept in the context history repository. The positions are expressed in terms of latitude and longitude as this is the positioning format provided by the GPS systems. As a result, the Cartesian coordinates of the cars' positions cannot be obtained but only their distances between each other. The distance history is then used as a reference point to infer whether a given distance indicates that the other car is ahead or behind. For instance, at time t_0 the car *a* is in position P_0. Later on at time t_1 the car *a* is in position P_1. At this time this car receives an emergency stop signal from car *b* in position P_2. It happens that the distance between P_1 and P_2 is d_0. However, this information does not tell us whether the car braking is ahead or behind. Hence, we obtain that distance between P_0 and P_2 is d_1 and that the reference distance between P_0 and P_1 is d_{ref}. If the distance d_{ref} is greater than the distance d_1, then car *a* is ahead car *b* otherwise the car *b* is ahead.

As a result of receiving an emergency stop signal, the brake control SO will generate the events "releaseBrake", "emergencyStop" and "releaseAcelerator" to the braking controller, the speed control SO and the network, respectively. Collisions with other cars are avoided by receiving a "carTooClose" event from the location information SO. In which case the events "brake" and "releaseAcelerator" are sent to the braking controller and the speed control SO, respectively. Regarding traffic light signals, when the "yellowSignal" or the "redSignal" are received, the same generated as for the car collision avoidance scenario. This also applies in the case or receiving the event "decreaseSpeed" from the QoS adaptation facility.

In case of detecting an obstacle or detecting a failure, the events “obstacleDetected” and “failureDetected” are received in which case the signals of the collision avoidance case are generated, as shown in table 5.3. Additionally the event “emergencyStop” is broadcasted to the nearby cars. Finally, the speed control SO controls the time at which the brake system is released by sending the event “releaseBreak”. As a result, the event “releaseBrake” will be sent to the braking controller.

Brake Control Sentient Object			
Input Events		Output Events	
Source	**Event type**	**Sink**	**Event type**
Speed Control SO	releaseBrake	Braking Controller	releaseBrake
network	emergencyStop	Braking Controller	emergencyStop
		Speed Control SO	releaseAcelerator
		network	emergencyStop
Location info SO	carTooClose	Braking Controller	brake
		Speed Control SO	releaseAcelerator
network	yellowSignal	Braking Controller	brake
		Speed Control SO	releaseAcelerator
network	redSignal	Braking Controller	brake
		Speed Control SO	releaseAcelerator
sensor	obstacleDetected	Braking Controller	emergencyStop
		Speed Control SO	releaseAcelerator
		network	emergencyStop
TCB	failureDectected	Braking Controller	emergencyStop
		Speed Control SO	releaseAcelerator
		network	emergencyStop
Coverage Stability	decreaseSpeed	Braking Controller	brake
		Speed Control SO	releaseAcelerator

Table 5.3. Input and Output events of the Brake Control Sentient Object

The speed control sentient object is basically in charge of making sure that the speed of the car is controlled in such a way that collisions are avoided and traffic lights obeyed, as shown in table 5.4. The event “speedup” is sent by the application to start moving the car. The speed control SO receives the events “speedup”, “carTooFar”, “greenSignal” and “speedUp” from the application, the location information SO, the other cars and the coverage stability facility, respectively. As a result, the events “releaseBrake” and “speedup” are sent to the brake control SO and the speed controller, respectively. When a car reaches maximum safe distance from a car ahead, the location information SO sends the “maintainSpeed” event. Lastly, the accelerators is released by the speed controller if the “realeaseAcelerator” event is received from the brake control SO.

In addition, the location information SO is responsible for managing the cars’ positioning information, as shown in table 5.5. More concretely, the event “carPosition” is received from the nearby cars. The location information SO stores the car position in the context history and either the event “carTooClose” or the event “carTooFar” is generated if the control engine decides so after taking into account its current position. These events are received by the brake control SO and the speed control SO, respectively. Similarly, in case of receiving the event “myPosition” from the GPS, the location SO additionally broadcast its current position to the nearby cars. Lastly, in case of receiving the event “trafficLightPosition”, both the position of the traffic light and its proximity group’s radius are stored in the context history.

When the car reaches the border of the proximity group, the car subscribes to the proximity group. As a result, the traffic light sends its signals to this car. The car then unsubscribes to this group when it is leaving the proximity area. The reason for cars to explicitly subscribe and unsubscribe to the proximity group is that no current underlying support is provided for making use of such groups.

Speed Control Sentient Object			
Input Events		Output Events	
Source	**Event type**	**Sink**	**Event type**
Brake Control SO	releaseAcelerator	Speed Controller	releaseAcelerator
application	speedup	Brake Control SO	releaseBrake
		Speed Controller	Speedup
Location info SO	carTooFar	Brake Control SO	releaseBrake
		Speed Controller	Speedup
Location info SO	maintainSpeed	Speed Controller	maintainSpeed
network	greenSignal	Brake Control SO	releaseBrake
		Speed Controller	Speedup
Coverage Stability	speedUp	Brake Control SO	releaseBrake
		Speed Controller	Speedup

Table 5.4. Input and Output events of the Speed Control Sentient Object

The traffic light SO receives requests for subscribing and unsubscribing to the proximity group, as said before, as shown in table 5.6. This SO sends the events “yellowSignal”, “redSignal” and “greenSignal” to the members of the group.

It is worth mentioning that the events generated by each primitive SO are produced in the particular order they were mentioned.

Location Sentient Object			
Input Events		Output Events	
Source	**Event type**	**Sink**	**Event type**
GPS	myPosition	Brake Control SO	carTooClose
		Speed Control SO	carTooFar
		network	myPosition
network	carPosition	Brake Control SO	carTooClose
		Speed Control SO	carTooFar
network	trafficLightPosition	network	subscribeTrafficLight
		network	unsubscribeTrafficLight

Table 5.5. Input and Output events of the Location Information Sentient Object

Traffic Light Sentient Object			
Input Events		Output Events	
Source	**Event type**	**Sink**	**Event type**
network	subscribeTrafficLight	----------	----------
network	unsubscribeTrafficLight	----------	----------
		network	yellowSignal
		network	redSignal
		network	greenSignal

Table 5.6. Input and Output events of the Traffic Light Sentient Object

5.2 Design of the Room Demo

5.2.1 Technology Infrastructure

5.2.1.1 Innovative Interactions Laboratory (IIL)

The sentient room demo aims to build an intelligent environment using the sentient object paradigm. We have set the sentient room demo in a semi-public space – the Innovative Interactions Laboratory (IIL) in computing department at Lancaster University. It is divided into three parts, a living room, a small machine room, and a kitchen. The main body of the IIL is the living room with a size of 7m*5.25m, which is illustrated in Figure 5.7. The living room contains visual devices, e.g., cameras, plasma screens; audio devices, e.g., speakers, microphones; and conventional room devices, e.g., lights, air-conditioner. The machine room is about 3m*3.3m, which hides the computing facilities, such as network switches, routers, complex music system, and servers connecting to the plasma screens. The kitchen contains all the conventional utilities, such as cooker, fridge, microwave, coffee machine, etc, as well as a web camera hanging from the ceiling.

Figure 5.7 The semi-public living room of IIL

In order to offer personalized intelligent services, the identity of the person has to be recognized. Recent biometric recognition devices, e.g., iris scanner, provide a high-level of accuracy for identifying persons, and they can reliably control access to some top security areas. The recognition process is very simple: you just walk up to and look into the iris scanner, which will verify your personal iris data by comparing it with the samples captured during the enrollment process. At the entrance of the IIL, an iris scanner and a library card reader (Figure 5.8) are installed, which provide two possible ways to identify the person who enters the room.

Figure 5.8 Iris scanner and library card reader at the entrance of IIL

Since the sentient room demonstrator is intended to show the inference and learning capabilities of the sentient objects, we introduced hardware sensors for getting environmental parameters, e.g., sound level, light intensity, temperature, etc. The current architecture of the sentient object suggests that these environmental data can be fused to get high level contexts, which are in turn fed into the inference or learning engine. An example of context fusion is to guess how many people are in the room by analyzing the noise level and temperature in the room.

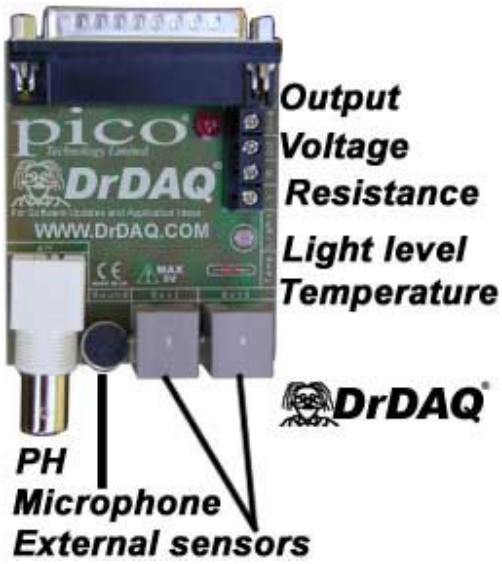


Figure 5.9 DrDAQ sensor board for gathering environment parameters

We have worked on some hardware sensor and finished wrapping it as a sensor object in CORTEX. The hardware sensor we had worked on is called DrDAQ sensor board [18], which has nine built-in sensors for light, sound (both sound level and sound waveforms) and temperature. It can also measure voltage and resistance if you connect the object with the

sensor board. In addition it has a socket for standard pH electrodes and two connectors for optional external sensors (temperature, humidity, pressure and more). This DrDAQ sensor board can be directly connected with the parallel port on the PC, from which it draws power so that no battery supplies are required.

5.2.1.2 Reflective Middleware

Refer to Section 5.1.1.

5.2.1.3 STEAM-based Event Channel

Refer to Section 5.1.2.1.

5.2.1.4 Preliminary Integration

We have chosen to model the hardware devices as sentient sensor and actuator software objects, e.g., the iris scanner, plasma screen, environmental sensor, etc. We have engineered an event channel based on the TCD's STEAM and built a preliminary prototype – the personalized homepage launching service. This service consists of the iris scanner sensor, the plasma screen actuator, and a sentient object performing reasoning in the middle. The prototype works as the follows: the iris scanner sensor produces a recognized event in XML format when someone enters the room; the sentient object consumes this event, maps the user identity that is extracted from the recognized event content to his/her homepage URL, and produces a display event; the plasma screen actuator consumes the display event and launches Internet explorer to the specific URL (this scenario could easily be expanded to incorporate other actuations, such as controlling physical room attributes: temperature, light etc.).

This demonstrator contains the core sensor, actuator, and sentient object components of the CORTEX programming model. A key component missing from our demonstrator is the inference engine or controlling logic in the sentient object. We intend to put the inference engine into the sentient object, and then refine our proof-of-concept prototype – a more sophisticated homepage launching service. The focus of the prototype is to develop the contextual reasoning component within our sentient object prototype, in order to 'give intelligence' to the room, so that it can decide how to display homepage when there are multiple persons coming to the room. The room can have different possible ways on how to display multiple persons' homepage: it can either display the homepage of the person who has the highest priority (the professor) or split the screen to display multiple homepages. No matter how the room decides to display the homepages, it has to have some decision making capability or intelligence, based on the limited contextual and sensor data available to it. The context component framework is also the focus of the research, and we can handle context information in a unified way by making use of it.

5.2.2 Design of Scenario

5.2.2.1 Sentient Object as Component Framework

A sentient object has been defined as "an entity that both consumes and produces software events, and lies in some control path between at least one sensor and actuator" [4]. A sentient object consists of three components: sensory capture, context representation, and inference engine. The architecture of the sentient object model is illustrated in figure 5.10.

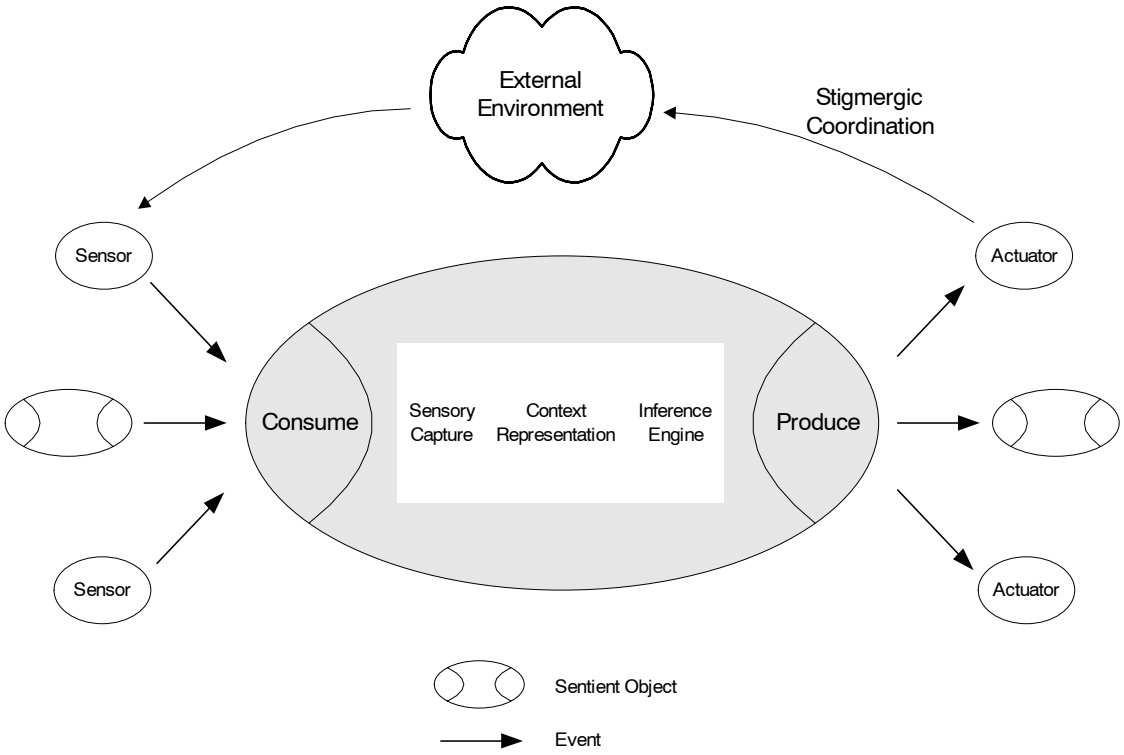


Figure 5.10 The Sentient Object Model Architecture

Getting contextual information from raw sensory data is the main task of context acquisition, and the major issues in the area of sensory capture are data filtering and sensor fusion. Dead-reckoning can be used if there is insufficient traffic or delays, and fusion can provide more accurate estimates of context. The context representation component deals with the representation of context information in a way that is useful for the sentient object and may be easily exchanged amongst sentient objects. A possible way of representing the context is to use XML, which is expressive, flexible, standard, and easy to handle. The inference component is actually the brain of a sentient object, and it has some form of decision making ability and intelligence. In order to make decision, some rules need to be applied on the current context. These rules can either be statically programmed into the engine or dynamically updated by learning the context history.

We are proposing to build a sentient object as a component framework, which makes use of publish/subscribe component framework and context component framework. The publish/subscribe component framework gives the sentient component framework the ability to consume and produce events. Context component framework provides a unified way for fusing, managing, inferring, and learning context across different applications.

Context Component Framework

The context component framework consists of four elements, which are either component or component framework. These elements are actually the counterparts of the components in the sentient object architecture. The context fusion component has the same functionality as the "context capture" component, and the inference engine components is an instance of the "inference engine" in the sentient object architecture. Since we are going to represent the context in XML format, we do not have a counterpart of the "context representation" component. Instead, we have a context management component that enables us to store,

update, query, or delete context. Learning CF is complementary to the inference engine component, and it can dynamically plug-in different machine learning algorithms that make the system learn the rules from previous context and interaction with the user. The learned rules can either be injected into the inference engine or used by the learning CF itself to make decision.

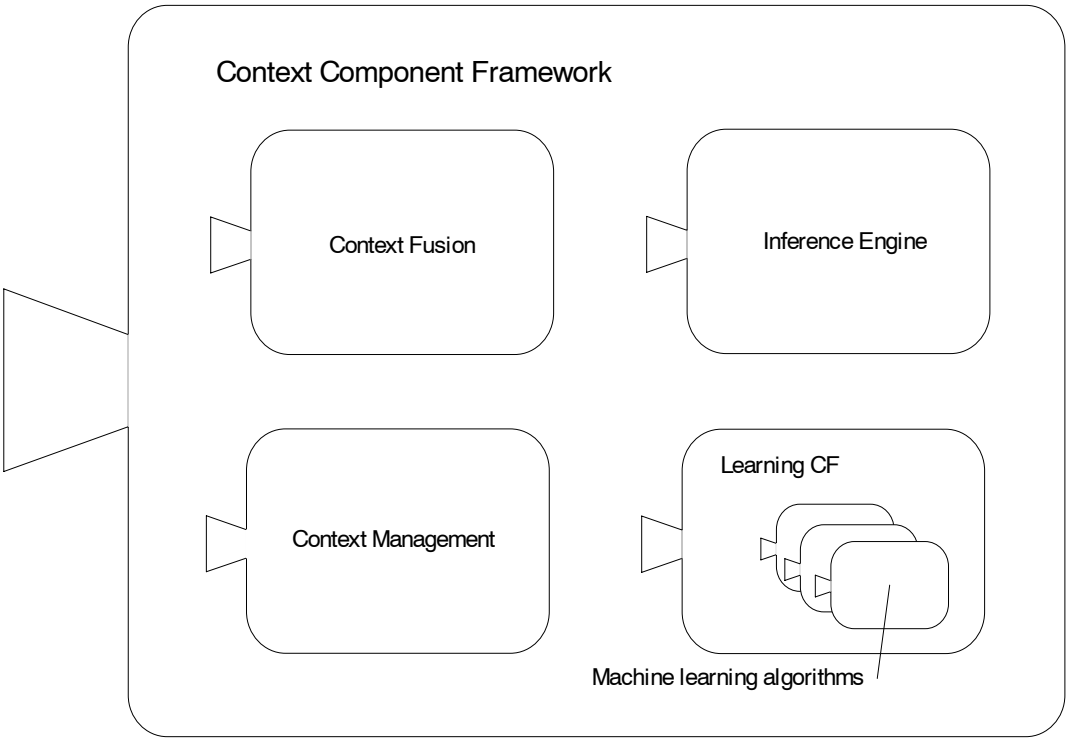


Figure 5.11 Context Component Framework

The context fusion component is responsible for filtering and fusing raw sensory data. Bayesian network method has been proposed by Trinity, and we might go for this method as the first step. Possible interfaces are as follows:

```
Interface IContextFusion : IUnknown {
      HRESULT ContextFiltering([in] SensoryData* RawData, [in] int
      accuracy, [out] SensoryData* FilteredData,);
      HRESULT CreateBayesianNetwork([in] unsigned char* name);
      HRESULT BayesianContextFusion([in] SensoryData** RawData, [out]
      SensoryData* FusedData, [out] int* accuracy);
}
```

Context Management Component

The context management component interacts with the context database, and it provides interfaces to store, update, query, or delete context as follows:

```
interface IContextManagment : IUnknown {
```

```
        HRESULT AddContext([in] ContextDatabase *cDb, [in] Context* c);
        HRESULT UpdateContext([in] ContextDatabase* cDb, [in] Context*
        c);
        HRESULT QueryContext([in] ContextDatabase* cDb, [in] String
        Query, [out] Context** c);
        HRESULT DeleteContext([in] ContextDatabase* cDb, [in] Context*
        c);
}
```

Inference Engine Component

Inference engine component interfaces with some artificial intelligence expert systems, e.g., CLISP, to infer high-level context from raw sensory data or control logic from context. We provide two methods for these two different purposes: “ContextInference” and “RuleInference”.

```
interface ICLISPInferenceEngine : IUnknown {
        HRESULT CreateCLISPInferenceEngine([in] unsigned char* name);
        HRESULT AddRule([in] unsigned char* name, [in] Rule* r);
        HRESULT UpdateRule([in] unsigned char* name, [in] Rule* r);
        HRESULT DeleteRule([in] unsigned char* name, [in] Rule* r);
        HRESULT ContextInference([in] SensorData** RawData, [out]
        Context* TargetContext, int* accuracy);
        HRESULT RuleInference([in] Context** InputContext, [out] Rule*
        InferredRule, int* accuracy);
        HRESULT RuleInference([in] Context** InputContext, [out]
        string* EventContent, int* accuracy);
}
```

Learning Component Framework

The learning component framework provides learning capabilities by generating rules form context history. This CF makes use of different machine learning algorithms, e.g., decision tree, artificial neural network, Bayesian leaning, etc, as its plug-ins so that it can choose the most appropriate candidate. Possible interfaces for the learning CF and its plug-ins are illustrated as follows:

```
interface IInferenceCF : IUnknown {
        HRESULT CreateLearningEngine([in] int Type, [out] IUnknown*
        LearningEngine);
        HRESULT Train ([in] unsigned char* ContextHistory);
        HRESULT TestContext ([in] Context* CurrentContext, [out] int*
        result);
        HRESULT TestContexts ([in] unsigned char* Contexts, [out] int*
        accuracy);
}
```

Decision tree learning is a method for approximating discrete-valued target functions, and it is robust to noisy data and capable of learning disjunctive expressions. The learned function is

represented in the form of a decision tree, which can also be described in some human friendly if-then rules.

```
interface IDecisionTree : IUnknown {
      HRESULT CreateDecisionTree([in] unsigned char* name);
      HRESULT GetInputSamples([in] unsigned char* filename);
      HRESULT Train ([in] unsigned char* ContextHistory);
      HRESULT TestContext ([in] Context* CurrentContext, [out] int*
      result);
      HRESULT TestContexts ([in] unsigned char* Contexts, int*
      accuracy);
      HRESULT DisplayDecisionTree ();
}
```

An artificial neural network is an information-processing paradigm inspired by the way the densely interconnected, parallel structure of the mammalian brain processes information. An artificial neural network is built out of a densely interconnected set of simple units (or nodes), where each unit takes a number of real-valued inputs (possibly outputs of other units) and produces a single real-valued output (which may become the input to many other units). The processing ability of the network is stored in the inter-unit connection strengths (or weights), obtained by a process of learning from a set of training patterns.

```
interface INeuralNetwork : IUnknown {
      HRESULT CreateNeuralNetwork([in] unsigned char* name);
      HRESULT GetInputSamples([in] unsigned char* filename);
      HRESULT Train ([in] unsigned char* ContextHistory);
      HRESULT TestContext ([in] Context* CurrentContext, [out] int*
      result);
      HRESULT TestContexts ([in] unsigned char* Contexts, int*
      accuracy);
}
```

5.2.2.1 Sentient Room as Sentient Objects

In the sentient office case study by Trinity, the sentient office has been modeled as one single sentient object. This super-large sentient object contains the control logic of all the devices in the office. The case study aimed to clarify the internal structure of a sentient object, but the design strategy is not suitable for a full-scale demonstration like the sentient room that will be built in Lancaster University. We need to bear in mind issues as scalability, extensibility, reusability, and customizability, all of which can in turn affect our design and implementation.

One of the strategic design decisions is to split the single super-large sentient object into a series of small-scale sentient objects, each of which maintains control logic for a certain application domain. For example, we have air-conditioner service sentient object that maintains the control logic of the air-conditioner to adjust appropriate temperature according to occupier's preference. We call sentient objects that maintain control logic of certain devices "service sentient objects", because they provide certain services to the end users. Other service sentient objects in the sentient room demo include homepage display service, news page display service, music playing service, etc.

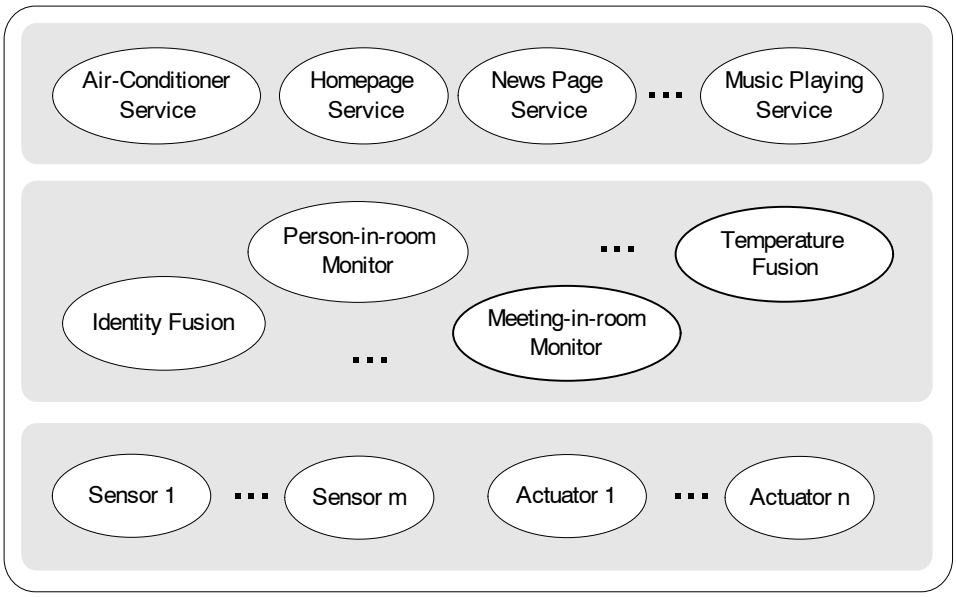


Figure 5.12 Software Objects in the Sentient Room

Moreover, some functions are actually so common that most of the room services need to make use of. For example, all of the service sentient objects need to know the identity of the person who is in the room in order to provide personalized services. In our case, we introduce "identity fusion" sentient object, which consumes software events from the iris scanner or library card reader actuator and produces high-level identity events to the service sentient objects. This sentient object just has fusion capability, so that we only need to insert the fusion component into the context framework. We name the sentient objects that provide common functions to high-level service sentient objects as "supporting sentient objects". Supporting sentient objects may only have fusion capability, but they can also possess inference or learning engine. For example, the person-in-room monitor produces events saying "how many persons are in the room" by inferring from the environmental parameters, such as temperature, noise level, light intensity, etc. Its inference rules might be learned from the context history. Figure 5.12 illustrates the software objects in the sentient room: sensors and actuators are at the bottom layer, supporting sentient objects are in the middle layer, and service sentient objects are at the top layer.

The context and the publish/subscribe component framework can provide most of the functionalities in the sentient object, so that we can start to build sentient room applications by making use of them. Figure 5.13 shows the architectural design of the sentient room, which consists of all kinds of software objects and event channels.

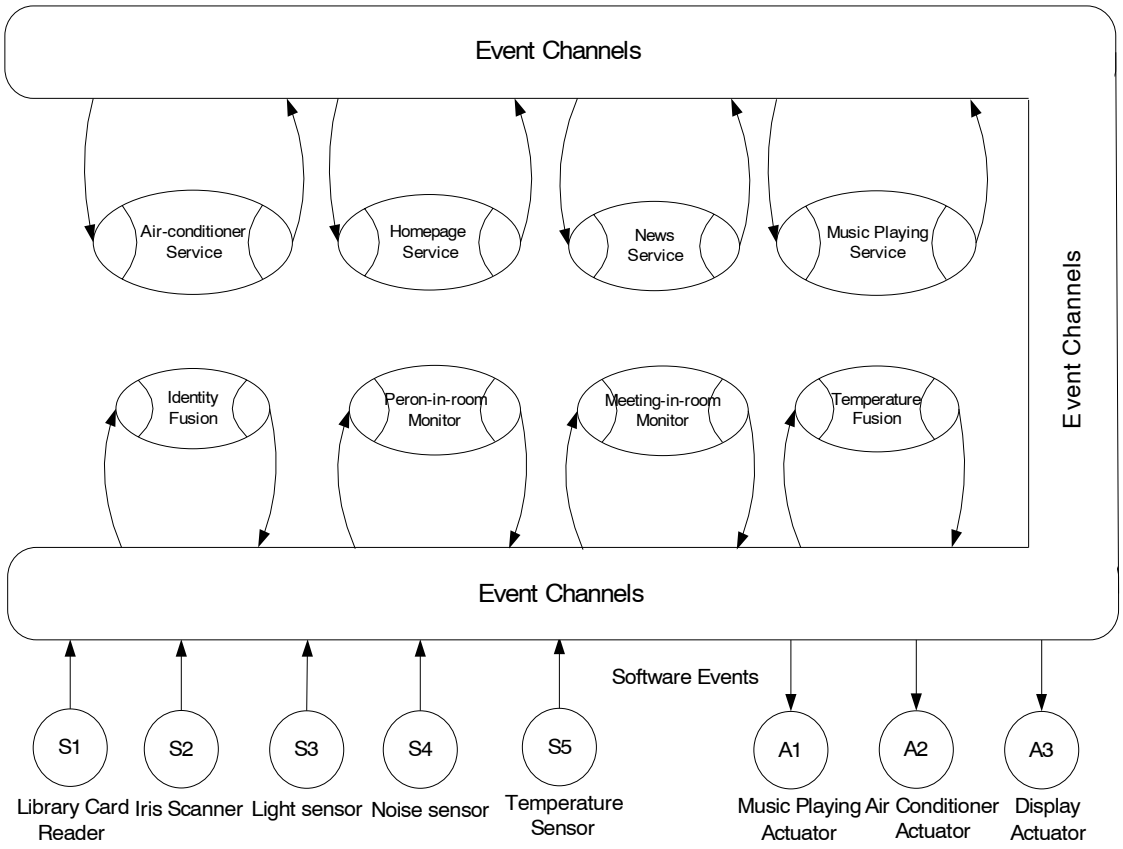


Figure 5.13 Sentient Room Software Architecture Design

The information of input and output events of the above sentient objects are illustrated in the tables below.

Supporting sentient objects

Identity Fusion Sentient Object			
Input Events		Output Events	
Source	**Event type**	**Sink**	**Event type**
Iris scanner sensor	ldrRecog	All service sentient objects	Recog
Library card reader sensor	irisRecog		

Person-in-room Sentient Object			
Input Events		Output Events	
Source	**Event type**	**Sink**	**Event type**
Light sensor	light	All service sentient objects	Person
Temperature sensor	temperature		
Noise sensor	noise		

Meeting-in-room Sentient Object			
Input Events		Output Events	
Source	**Event type**	**Sink**	**Event type**
Light sensor	light	All service sentient objects	Meeting
Temperature sensor	temperature		
Noise sensor	noise		

Temperature Fusion Sentient Object			
Input Events		Output Events	
Source	**Event type**	**Sink**	**Event type**
Temperature sensor	temperature	Air-conditioner service sentient objects	FusedTemp

Service sentient objects

Air-conditioner Service Sentient Object			
Input Events		Output Events	
Source	**Event type**	**Sink**	**Event type**
Identity fusion sentient object	Recog	Air-Conditioner Actuator	ChangeTemp
Person-in-room sentient object	Person		
Meeting-in-room sentient object	Meeting		
Temperature fusion sentient object	FusedTemp		

Homepage Service Sentient Object			
Input Events		Output Events	
Source	**Event type**	**Sink**	**Event type**
Identity fusion sentient object	Recog	Display Actuator	Display
Person-in-room sentient object	Person		
Meeting-in-room sentient object	Meeting		

News Service Sentient Object			
Input Events		Output Events	
Source	**Event type**	**Sink**	**Event type**
Identity fusion sentient object	Recog	Display Actuator	Display
Person-in-room sentient object	Person		
Meeting-in-room sentient object	Meeting		

Music Playing Service Sentient Object			
Input Events		Output Events	
Source	**Event type**	**Sink**	**Event type**
Identity fusion sentient object	Recog	Music Playing Actuator	PlayMusic
Person-in-room sentient object	Person		
Meeting-in-room sentient object	Meeting		

6. Conclusions

We have revised the main characteristics of CORTEX applications. It has been mentioned that these applications commonly cover a part of the large spectrum defined by the CORTEX characteristics. Therefore, two applications that altogether cover all the characteristics were identified, namely, a car control system and a smart room scenario. The former is focused on time- and safety-critical systems in mobile ad hoc environments. In contrast, the latter emphasis the use of intelligent behaviour to solve conflicts in user preferences and learning capabilities to automatically capture the new preferences and habits of a user.

In addition, a middleware architecture was presented which has three main roles in the demonstrator. Firstly, the middleware functions as an integrator which joins the various contributions of the different project partners in a single system architecture. Secondly, the middleware is used as a means to reuse the architecture by the tow demo applications rather than implementing a single solution for each demo. Lastly, the middleware is employed to support both configurability and reconfigurability capabilities when unexpected changes are detected in the environment.

Finally, we have presented the designs of the demo scenarios which encompass both the technology infrastructure and details about the design of each scenario. The technology infrastructure for the car control system includes a number of car robots controlled by an iPAQ. The car is also provided with a wireless network card, a number of ultrasound sensors and a GPS. The room scenario then counts with the Interaction Lab supplied with an iris scanner, a number of sensors, a big plasma screen, etc. In addition, the design of the demo applications identified the sentient objects that constitute the systems. Lastly, the input events and the output events of each sentient object were defined.

Appendix A

Publish Subscribe Service Interfaces

This section describes the semantic behavior of the interfaces which make up the publish/subscribe Service. Interface (IDL) of each component is presented, along with a brief description of the purpose of the component. For each interface in the components, a brief description of its purpose is provided, along with an explanation of the semantics of each of its operations and attributes. The interfaces are specified in Microsoft interface definition language (MIDL). The interfaces defined are for the preliminary version of publish/subscribe service, and it will be extended in light of advanced support for Quality of services. The components are built using OpenCOM.

The publish/subscribe service is defined in terms of the following components:

Event Channel Factory Component:

This component defines the interface for creating event channels for publishing behavior and subscribing behavior instances.

Publisher Component:

This component defines the publisher interface for basic publisher communication.

Subscriber Component:

This component defines Subscriber interface for the basic consumer communication.

Filter Component:

This component defines the interface for filters and event channel properties supported by the publish/subscribe service.

Application Notify Component:

This component defines the interface, which has the call-back function to notify subscribing entities about new event notifications.

Dispatcher Component:

This component defines the interface for controlling dispatching events to subscribers.

SOAP Messaging Component:

This component defines the interface for SOAP messaging protocol.

SOAP-to-Multicast Component:

This component defines the interface for binding SOAP-messaging protocol to multicast protocol.

Multicast Component:

This component defines the interface for multicast protocols.

Each of the interfaces corresponding to the components is defined in its own subsection as follows.

Event Channel Factory Component

The EventChannelFactory interface defines methods for creating and managing new event channels. It defines routines that create new instances of event channels with publisher style and subscriber styles. More specifically, it defines methods for creating publisher side components configuration and subscriber side components configuration instances. The EventChannelFactory interface is defined as follows

```
//Forward declarations
interface  IPublish;
interface  IFilter;
interface  ISubscribe
interface  IApplicationNotify;

typedef struct PUBLISHER_API
{    IPublish* pIPublish;
     IFilter*  pIFilter;
} PUBLISHER_API;


typedef struct SUBSCRIBER_API
{    ISubscribe* pISubscribe;
     IFilter*    pFilter;
     IApplicationNotify* pIApplicationNotify;;
} SUBSCRIBER_API;

interface  IEventChannelFactory : IUnknown {

     PUBLISHER_API* Create_Channel_Publisher();
     SUBSCRIBER_API* Create_Channel_Subscriber();
}
```

The *Create_Channel_Publisher* method creates an instance of publisher side components, configures the components configuration and returns interface pointers to IPublish interface and IFilter interface. In other words, this method (or operation) creates an instance of an event channel which has a publishing style.

The *Create_Channel_Subscriber* method creates an instance of subscriber side components, configures the components configuration and returns interface pointers to ISubscribe interface, IFilter interface and IApplicationNotify interface. In other words, *Create_Channel_Subscriber* method creates an instance of an event channel which has a subscribing style.

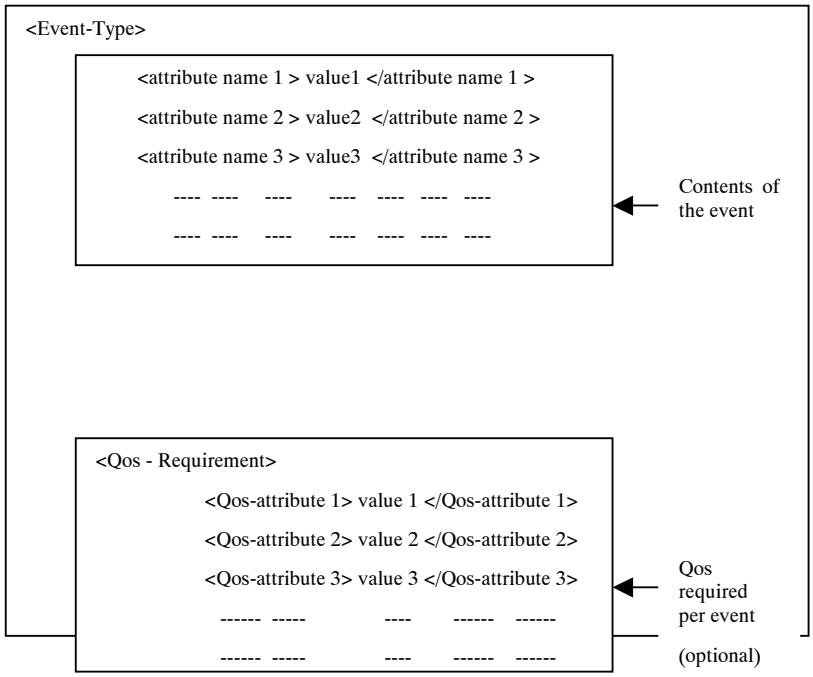
The Quality of service (QoS) properties of event channels can be set using IFilter interface pointers returned by the above two calls. The IFilter interface has methods to specify functional and non functional QoS properties of the event channel instantiated. The methods of IFilter interface is looked into in the Filter component section.

Publisher Component

The component has the IPublish interface, defined as follows

```
interface  IPublish : IUnknown {
            HRESULT Publish([in] char* xmlEvent);
}
```

The publisher application constructs an XML based event, which specifies event type, event attributes –value pairs and Qos requirements per event basis. The instance of the XML based event constructed should conform to the generic XML profile defined as follows.



The XML profile defined to represent events is extensible, as there are no restrictions on the number of event attribute- value pairs. The QoS attribute specification is also extensible. The event value, data types supported are string, integer and float. The QoS can be specified on per event basis as required, such as priority, deadline, reliability, expiration etc. However, per event QoS specification is not the only level in which QoS can be specified. Once the event is created it can be published using the publish method, to all interested subscribers via the respective event channel. The publisher application gets the pointer to IPublish interface, from the call to *Create_Channel_Publisher* method as previously noted. Moreover, the publisher application programmer should set the event channels' functional and non functional properties, using the IFilter interface, before invoking any publish call. Details of IFilter interface follows later. The implementation of the IPublish interface can additionally include code, to piggyback context information, such as location of publishing entity, to the original event published. By this way, context information can be transparently added the original event.

Subscriber Component

The subscriber component has the ISubscribe interface, defined as follows

```
interface ISubscribe : IUnknown {
          HRESULT Subscribe();
          HRESULT UnSubscribe();
}
```

The application programmer obtains the pointer to ISubscribe interface, by calling *Create_Channel_Subscriber* method on the IEventChannelFactory interface. Application programmer should set the functional, non functional properties of the event channel and the subscription, before invoking any methods on the ISubsribe interface. The subscribe method starts the subscription, where the subscriber starts to get notified of events (via call-back function) which match the subscription. The unsubscribe method cancels the current subscription. The application programmer should implement the call-back function according to his/her preference. The detail of the call-back function is given in ApplicationNotify component section.

Filter component

The filter component is in charge of filtering events, at both, the publisher side and the subscriber side. The filter component supports subject, content and context based event filtering. Moreover, the filter component is in charge of setting up the event channel properties. The interface of filter component is defined as follows.

```
interface IFilter : IUnknown {
        HRESULT   set_filter([in] char* FEL_filter);
        char*     get_filter();
        char*     set_Qos([in] char* xmlQos);
        Char*     get_Qos();
        Char*     validate_QoS([in] char* xmlQos);
        boolean   filter([in] char* xmlEvent);

}
```

The application programmer can define the subscription by calling the *set_filter* method and by passing the filter expression defined in FEL, FEL is a simple filtering constraint grammar. FEL supports subject and content based event filtering. It is extended to support context based event filtering such as distance.

Filters are supported in publish/subscribe service as components which can be associated with event channels at publisher side and subscriber side. Each filter component has associated with it constraints which have meaning in the particular filtering constraint grammar.

The *set_qos* method takes as an input parameter a set of name-value pairs, represented in XML, which encapsulates quality of service property settings that a client is requesting, the target event channel. If the implementation of the target event channel is not capable of supporting any of the requested quality of service settings, or if any of the requested settings would be in conflict with a QoS property defined at another level; the return value specifies the QoS parameters not supported. The return value contains data sequence represented in XML, which identifies the name of a QoS property in the input list whose requested setting could not be satisfied.

The *validate_qos* method, accepts as input, a sequence of QoS property name-value pairs, represented in XML, which specify a set of QoS settings that a client would like to know if the target event channel is capable of supporting. If the any of the requested settings could not be satisfied by the target event channel, the method returns this information, a data sequence as XML structures, each of which identifies the name of a QoS property in the input list whose requested setting could not be satisfied.

The filter method, matches events represented in XML against filters defined in the filtering constraint grammar, and returns, specifying whether there is a match or not.

Application Notify Component

This component has the call-back method (or function), which is invoked when a new matching event is received at the subscriber. It should be noted, that, this is the *only* component which the application programmer has to implement. The interface of Application Notify component is defined as follows.

```
interface IApplicationNotify : IUnknown {
        HRESULT   call_back([in] char* xmlEvent);
}
```

The application programmer may add more methods to the IApplicationNotify interface as required to support the notification behavior.

Dispatcher Component

This component dispatches events received from the network layer, to the higher layer. This component controls event dispatching, event queuing, and prioritization of events according to their QoS requirements. The interface of Dispatcher component is defined as follows

```
interface IDispatcher: IUnknown
{
      HRESULT DispatchEvent([in] char* xmlEvent);
};
```

SOAP Messaging Component

This component exports the interface ISOAP_Messaging. This component supports SOAP messaging. The SOAP messaging component can bind to a synchronous transport such as HTTP, RPC or asynchronous transport such as multicast and broadcast. Since we use multicast protocol, the SOAP messaging utilises multicast protocol to do the messaging. The services of SOAP Messaging component are utilised by subscriber and publisher components to send and receive SOAP messages.

The interface ISOAP_Messaging is defined as follows :

```
interface ISOAP_Messaging : IUnknown {

   HRESULT SetSOAPEnvironment(char* URI);
   char* GetSOAPEnvironment();
   HRESULT SetSOAPEncoding(char* URI);
   char* GetSOAPEncoding();
   HRESULT SetURI(char* URI);
   char* GetURI();
   HRESULT SetURL(char* URI);
   char* GetURL();
   HRESULT SetStyle(int Sync);

   HRESULT SetToReceive();
   HRESULT SendSOAPMessage([in] char* XML);
   HRESULT ReceiveSOAPResponse([out] char* Message);
   HRESULT ReceiveSOAPMessage([out]char* Message,[out]char* url);
   HRESULT SendSOAPResponse([in] char* XML);

   int GetFaultCode();
   char* GetFaultString();
}
```

The details of methods of ISOAP_Messaging interface is not further discussed as it concerns SOAP messaging protocol.

SOAP-to-Multicast Component

This component exports the interface ISOAPTransport. This component binds the SOAP_messaging protocol to the multicast protocol. The interface of SOAP-to-Multicast component is defined as follows.

```
interface ISOAPTransport: IUnknown {

   HRESULT  Receive();
   HRESULT  SendSOAPMessage(unsigned char* URL, int Size,unsigned
                    char* SOAPEnv, [out]int* tsap, [in] int Sync);
  HRESULT ReceiveSOAPResponse([out] char* Resp, int tsap);
  HRESULT ReceiveSOAPMessage([out] int* tsap, [out]  unsigned char*
                              Message, [out] char* url);
  HRESULT SendSOAPResponse(unsigned char* SOAPEnv, int tsap);
  HRESULT StopReceive();
};
```

The details of methods of `ISOAP_Transport` interface is not further discussed as it concerns binding SOAP messaging protocol to multicast protocol.

Multicast Component

Multicast component has the following interface

```
Interface IMulticast : IUnknown {

        HRESULT     Send([in] char* xmlEvent);
        HRESULT     Receive([out] char* xmlEvent);
```

The *JoinGroup* method joins the multicast group associated with the event type specified in the input parameter. The *LeaveGroup* method leaves the respective multicast group. The *send* method, multicasts the event message to the respective multicast group. The *receive* method *receives* event messages from the corresponding group.

Application Programmer Interface

The API which the application programmer uses is listed below:

```
PUBLISHER_API*      create_channel_publisher();
SUBSCRIBER_API*     create_channel_subscriber();
HRESULT             set_filter([in] char* FEL_filter);
char*               get_filter();
char*               set_Qos([in] char* xmlQos);
Char*               get_Qos();
Char*               validate_QoS([in] char* xmlQos);
HRESULT             publish([in] char* xmlEvent);
HRESULT             subscribe();
HRESULT             unSubscribe();
```

Appendix B

The Service Discovery Framework

The Service Discovery framework allows services that have been advertised by different service discovery protocols to be found. The component configuration is configured to the discovery technology currently used in the environment. For example, if only SLP is currently in use, the framework's configuration will be an SLP Lookup personality. However, if SLP and UPnP are both being utilised at a location then the framework's configuration will include component implementations to discover both.. An application may also require services to be advertised; therefore, the personality can be changed to include service registration functionality using one or more protocols of choice. As in the Binding CF, the framework allows individual components to be changed, added or deleted. This is beneficial due to the range of functionalities that service discovery technologies offer. For example, in SLP you may wish to perform lookup using just the multicast protocol if no directory agent is present, but at a later stage if a directory agent is discovered the configuration can be changed to direct requests to it, rather than send multicast requests.

The service discovery framework offers a set of generic service discovery methods through the IServiceLookup Interface described below. This includes a generic service lookup operation that returns the information from different service discovery protocol searches in a generic format. For example, a lookup of a weather service across two discovery configurations, e.g. UPnP and SLP, returns a list of matched services from both types. It is this information (the description of the service returned by the lookup protocol) that is used to configure the binding framework.

However, the discovery protocol(s) that are currently in use at a location must be determined. Therefore, the service discovery framework provides two styles of functionality to find whether particular types of service discovery protocols are in use. The DiscoverDiscoveryProtocol component is plugged into the framework that tests if individual service discovery protocols are in use, either upon a synchronous request or by continuously monitoring the environment and generating an event on detection. Continuous monitoring will quickly use up resources (e.g. battery power); therefore in some cases synchronous checking may be appropriate. The top-level ReMMoC component framework utilises this behaviour to automatically reconfigure the service discovery framework based upon monitoring of the environment. The operations defined in the IDL definition of the IDiscoverDiscoveryProtocol interface illustrated below provide this functionality.

Other methods for discovering discovery protocols, not currently included in the implementation, may utilise the device's context information, e.g. if the device is currently using a Bluetooth connection then an SDP personality is configured. Furthermore, the middleware may use prior knowledge to select an appropriate protocol, i.e. the platform stores context information per location that details which service discovery protocols were used at that point previously.

We have implemented the service discovery framework with two service discovery protocol implementations: SLP and UPnP both with service lookup and registration capabilities, allowing us to demonstrate how to overcome the problems of the availability of multiple service discovery protocols. However, as with the binding framework, it is feasible for new discovery protocols to be integrated into the framework.

```
interface IServiceLookup : IUnknown {
    HRESULT ServicesFind([in] char* ServiceType, [in] int TimeToSearch, [in]

ReMMoCServiceFindHandler cback,);
    HRESULT GetAttributes([in] char* ServiceID, [out] AttributeList list);
}


interface ReMMoC_IDiscoverDiscoveryProtocol : IUnknown {
    HRESULT AsynchronousDiscoveryProtocolSearch([in] ServiceDiscoveryType
list[], [in] int
TimeToSearch, int length, [in] ReMMoCServiceFindHandler cback);
    HRESULT SynchronousDiscoveryProtocolSearch([in] ServiceDiscoveryType
sdt);
}
```

Appendix C

Resource management API

All component types support an interface with operations to transverse their associated abstraction hierarchies: `getLL()`, `setLL()`, `getHL()` and `setHL()`. For instance, the resource hierarchy may be traversed by applying the `getLL()` operation at the top-level, i.e. the VTM. This operation would be later applied to the lower-level resources, and so on. Both the higher-level and the lower-level of an entity in the hierarchies may be set by accessing the operations `setHL()` and `setLL()` respectively. The Access to both the manager and factory of a passive resource can be obtained through the interface `Iresource`, as shown below. The references to the manager and factory of an abstract resource are registered by the operations `setManager()` and `setFactory()` respectively.

```
interface IResource : IUnknown
{
        HRESULT getLL([out, size_is(,*maxRes)] IResource**, [out] long* maxRes);
        HRESULT setLL([in, size_is(,maxRes)] IResource**, [in] long maxRes);
        HRESULT getHL([out] IResource**, [out] IJob**);
        HRESULT setHL([in] IResource*, [in] IJob*);
        HRESULT getManager([out] IManager**);
        HRESULT setManager([in] IManager*);
        HRESULT getFactory([out] IResourceFactory**);
        HRESULT setFactory([in] IResourceFactory*);
        };
```

The interface of a job includes the operations `getSchedParam()` and `setSchedParam()`. The former is in charge of accessing predefined settings. The latter is responsible for performing a control admission test. If successful, resources are reserved and the scheduling parameters are set. The operation `run()` allows for the execution of a function with associated parameters. Jobs can also be suspended by using the operation `suspend()` and `resume()` respectively.

```
interface IJob : IUnknown
{
        HRESULT getLL([out, size_is(,*maxRes)] IResource**, [out] long* maxRes,
                [out, size_is(,*maxJob)] IJob**, [out] long* maxJob);
        HRESULT setLL([in, size_is(,maxRes)] IResource**, [in] long maxRes,
                [in, size_is(,maxJob)] IJob**, [in] long maxJob);
        HRESULT getHL([out] IJob**);
        HRESULT setHL([in] IJob*);
        HRESULT getManager([out] IScheduler**);
        HRESULT setManager([in] IScheduler*);
        HRESULT getFactory([out] IJobFactory**);
        HRESULT setFactory([in] IJobFactory*);
        HRESULT GetSchedParam([out] OLECHAR* schedParam);
        HRESULT SetSchedParam([in] OLECHAR* schedParam);
        HRESULT run([in] void* function, [in] void* parameters);
        HRESULT suspend();
        HRESULT resume();
};
```

The interfaces of factories expose an operation for the creation of abstract resources, as shown below. This operation is also responsible for associating the resource with a resource manager. In case of creating processing resources, a job factory is used and the scheduling parameters should be indicated. This interface also provides the operation `getResources()` that returns references of the resources created by the factory.

```
interface IResourceFactory : IUnknown
{
        HRESULT getLL([out, size_is(,*maxRes)] IResourceFactory**, [out] long* maxRes);
        HRESULT setLL([in, size_is(,maxRes)] IResourceFactory**, [in] long maxRes);
        HRESULT getHL([out] IResourceFactory**, [out] IJobFactory**);
        HRESULT setHL([in] IResourceFactory**, [in] IJobFactory**);
        HRESULT newResource([in] int size, [in] OLECHAR* policy);
        HRESULT getResources([out, size_is(,*maxRes)] IResource**, [out] long* maxRes);
};

interface IJobFactory : IUnknown
{
        HRESULT getLL([out, size_is(,*maxRes)] IResourceFactory**, [out] long* maxRes,
                [out, size_is(,*maxJob)] IJobFactory**, [out] long* maxJob);
        HRESULT setLL([in, size_is(,*maxRes)] IResourceFactory**, [in] long maxRes,
                [in, size_is(,maxJob)] IJobFactory**, [in] long maxJob);
        HRESULT getHL([out] IJobFactory**);
        HRESULT setHL([in] IJobFactory*);
        HRESULT  newResource([in]  int  size,  [in]  OLECHAR*  policy,  [in]  OLECHAR*
schedParam,
                [out] IJob** interf);
        HRESULT  getResources([out,  size_is(,*maxJob)]  IJob**  interf,  [out]  long*
maxJob);
};
```

In addition, the VTM factory should implement the interface `IVtmFactory` which offers operations for obtaining the associated task of a VTM and vice versa, i.e. the IJob interface of a VTM. An operation for obtaining the interface `Ischeduler` of the VTM scheduler is also provided.

```
interface IVtmFactory : IUnknown
{
        HRESULT getVtm([in] OLECHAR* task_name, [out] IJob** vtm);
        HRESULT getTask([in] IJob* vtm, [out] OLECHAR* task_name);
        HRESULT getVtmScheduler([out] IScheduler** vtmScheduler);
};
```

The manager's interface exposes the operation `admit()` which performs an admission control test that determines whether or not there are enough resources to satisfy a resource request. In a successful case, resources may be reserved by using the operation `reserve()`. Reservations can then be liberated by invoking the operation `expel()`. These three operations are delegated to the policy component. Similar to factories, through the operation `getResources()`, resource managers are able to retrieve the references of the resources that are mapped or multiplexed. Such references may be included or removed from a manager's registry by using the operations `addResource()` and `removeResource()` respectively. In addition, the management policy is obtained by accessing the operation `getPolicy()` whereas the operation `setPolicy()` allows the user to set the management policy of the manager, i.e. the management policy component is dynamically changed.

```
interface IManager : IUnknown
{
        HRESULT getLL([out, size_is(,*maxMgrs)] IManager**, [out] long* maxMgrs);
        HRESULT setLL([in, size_is(,*maxRes)] IResource**, [in] long maxRes);
        HRESULT getHL([out] IResource**, [out] IJob**);
        HRESULT setHL([in] IResource*, [in] IJob*);
        HRESULT getPolicy([out] OLECHAR** policy);
        HRESULT setPolicy([in] OLECHAR* policy);
        HRESULT admit([in] OLECHAR* resourceAmount);//may define one or more parameters
        HRESULT reserve([in] OLECHAR* resourceAmount);
        HRESULT expel([in] OLECHAR* resourceAmount);
        HRESULT getResources([out, size_is(,*maxRes)] IResource**, [out] long* maxRes);
        HRESULT addResource([in] IResource**);
        HRESULT removeResource([in] IResource**);
};
```

The scheduler's interface provides similar operations to the manager's but additionally include operations for suspending and resuming processing resources by invoking the `suspend()` and `resume()` operations respectively. Lastly, the order of execution of multiplexed resources is determined by the `schedule()` operation which is also delegated to the policy component.

```
interface IScheduler : IUnknown
{
        HRESULT getLL([out, size_is(,*maxMgr)] IManager**, [out] long* maxMgr,
                [out, size_is(,*maxSched)] IScheduler**, [out] long* maxSched);
        HRESULT setLL([in, size_is(,maxMgr)] IManager**, [in] long maxMgr,
                [in, size_is(,maxSched)] IScheduler**, [in] long maxSched);
        HRESULT getHL([out] IScheduler**);
        HRESULT setHL([in] IScheduler*);
        HRESULT getPolicy([out] OLECHAR** policy);
        HRESULT setPolicy([in] OLECHAR* policy);
        HRESULT admit([in] OLECHAR* resourceAmount);//may define one or more parameters
        HRESULT reserve([in] OLECHAR* resourceAmount);
        HRESULT expel([in] OLECHAR* resourceAmount);
        HRESULT getResources([out, size_is(,*maxJob)] IJob**, [out] long* maxJob);
        HRESULT addResource([in] IJob*);
        HRESULT removeResource([in] IJob*);
        HRESULT suspend([in] IJob*);
        HRESULT resume([in] IJob*);
        HRESULT schedule([in] int quantum);
};
```

Policy components are in charge of performing control admission tests, resource reservation and resource liberation. An operation for obtaining the policy deployed by the component is also provided.

```
interface IManagementPolicy : IUnknown
{
        HRESULT getPolicy([out] OLECHAR** policy);
        HRESULT admit([in] OLECHAR* resourceAmount);
        HRESULT reserve([in] OLECHAR* resourceAmount);
        HRESULT expel([in] OLECHAR* resourceAmount);

};
```

Scheduling policy components additionally offer operations for scheduling jobs and the `dispatch()` operation for obtaining the next job to be executed.

```
interface ISchedulingPolicy : IManagementPolicy
{
        HRESULT schedule([in] int quantum, [in, size_is(,maxJob)] IJob**,
                [in] long maxJob);
        HRESULT dispatch([out] IJob**);
};
```

Appendix D

TCB API

TCB_Services() Creates an interface to TCB services

int close() Closes a TCB session.

LocalFDInfo endLocalFD(int tag)
Calls the tcb service: end_local_fd.

MeasurementInfo endLocalMeasurement(int tag)
Calls the tcb service: end_measurement.

int getGlobalTimestamp()
Calls the tcb service: get_global_timestamp.

int getGlobalTimestamp(int localInstant)
Calls the tcb service: get_global_timestamp
Returns the global clock value (milliseconds) at
local instant specified.

char getTCBId() Calls the tcb pseudo-service: get_tcb_id.

int getTimestamp() Calls the tcb service: get_timestamp.

int open() Opens a TCB session.

int startLocalFD(int start_ev, int spec,
int deadline, String dllName,
String funcName, String wcetName)
Calls the tcb service: start_local_fd.

int startLocalMeasurement(int start_ev)
Calls the tcb service: start_measurement.
int timelyExec(int start_ev, int delay,
int max_exec, String dllName,
String funcName, String wcetName)
Calls the tcb service: timely_exec.

TCBSocket(InetAddress ia, int port, TCB_Services tcbs)
Creates a TCBSocket, bound to the specified local address.

TCBSocket(InetAddress ia, int port, TCB_Services tcbs, String dllName,
String funcName, String wcetName, int deadline)
Creates a TCBSocket, bound to the specified local
address; specifying simultaneously the handler to be executed if
a remote distributed timing failure occurs.

void close() Close this socket.

ReceiveInfo endDistributedAction(DistributedFDId dfi)
Ends a distributed failure detection.

DistributedActionResult receive(TCBDatagramPacket tcbdp)
Receives a TCBDatagramPacket from this socket.

void send(TCBDatagramPacket tcbdp, int sendEv)
Sends a TCBDatagramPacket from this socket, starting a distributed measurement.

DistributedFDId sendWithFD(TCBDatagramPacket tcbdp, char destMask, int sendEv, int spec, int deadline, String dllName, String funcName, String wcetName)
Sends a TCBDatagramPacket from this socket, starting simultaneously a distributed measurement and a distributed timing failure detection; if a timing failure occurs, the failure handler is executed in the sender.

DistributedFDId sendWithRemoteFD(TCBDatagramPacket tcbdp, char destMask, int sendEv, int spec)
Sends a TCBDatagramPacket from this socket, starting simultaneously a distributed measurement and a distributed timing failure detection; if a timing failure occurs, the failure handler is executed in the recipients.

DistributedFDInfo waitInfo()
Waits for info from TCB distributed timing failure detection service.

References

[1] CORTEX. "Preliminary Definition of CORTEX programming Model."CORTEX deliverable D2 version 1.0, March 2002.

[2] Duran-Limon, H. A., G. S. Blair, et al. "Resource Management for the Real-Time Support of an Embedded Publish/Subscribe System." *Technical Report MPG-03-02.* 2003.

[3] Paul Grace, G. B. "Interoperating with Services in a Mobile Environmet." *Accepted in Proc. ACM/IFIP International Middleware Conference (Middleware'2003)*, Rio de Janeiro, Brazil. June 2003.

[4] "Preliminary Specification of Basic Services and Protocols."CORTEX Project. IST-2000-26031. Deliverable D5, February 2003.

[5] "MIT Project Oxygen." http://oxygen.lcs.mit.edu/

[6] "Sentient Computing Project at AT&T Laboratory." http://www.uk.research.att.com/spirit/

[7] Brumitt, B., B. Meyers, et al. “EasyLiving: Technologies for Intelligent environment.” *Handheld and Ubiquitous Computing* **September**: 2000.

[8] "The Aware Home at Georgia Institute of Technology." http://www.cc.gatech.edu/fce/ahri/.

[9] Fitzpatrick, A., G. Biegel, et al. "Towards a Sentient Object Model, Position Paper." *Workshop on Engineering Context-Aware Object Oriented Systems and Environments (ECOOSE)*, Seattle WA, USA. November 2002.

[10] Blair, G. S., G. Coulson, et al. “The Design and Implementation of Open ORB version 2.” *IEEE Distributed Systems Online Journal* **2**(6): 2001.

[11] Clarke, M., G. Coulson, et al. "An Efficient Component Model for the Construction of Adaptive Middleware." *IFIP/ACM Middleware'2001*, Heidelberg, Germany. November 2001.

[12] Microsoft Corporation. "COM: Delivering on the Promises of Component Technology." Microsoft Corporation. 2000. http://www.microsoft.com/com/default.asp

[13] Rene Meier, V. C. "Steam: Event-based Middleware for Wireless Ad Hoc Networks." *In Proceeding of the International Workshop on Distributed Event-Based Systems (ICDSC/DEBS'02)*, Vienna, Austria. 2002. pp. 639-644

[14] Sivaharan, T."Publish Subscribe Component-based Middleware for PDAs in Wireless Ad-hoc Networks." M. Sc., Lancaster University. 2002.

[15] W3C. "Simple Object Access Protocol (SOAP) 1.1.". 2000.

[16] Duran-Limon, H. A. and G. S. Blair. "Reconfiguration of Resources in Middleware." *Seventh IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2002)*, San Diego, CA. January 2002.

[17] Verissimo, P. and A. Casimiro. “The Timely Computing Base Model and Architecture.” *Transaction on Computers - Special Issue on Asynchronous Real-Time Systems* **51**(8): August 2002.

[18] "Homepage of the DrDAQ sensor board." http://www.drdaq.com/